THE UNIVERSITY OF NEW MEXICO

ECE 538
Advanced Computer Architecture

Instructor: Lei Yang

Department of Electrical and Computer Engineering

December 7, 2021

Final Exam Review

Basic concepts of
Parallel processing and Pipelining

Types of Parallelism

- Data Parallelism
  - Each processor performs the same task on different data
- Task Parallelism
  - Each processor performs different independent tasks
- Instruction Level Parallelism (ILP)
  - Multiple instructions are executed concurrently

SERIAL AND PARALLEL COMPUTATION

- Example-1: accountants adding the total amount of invoices

Add the total amount in m=1024 invoices as quickly as possible by n accountants. Find the time (T) required to add these numbers?

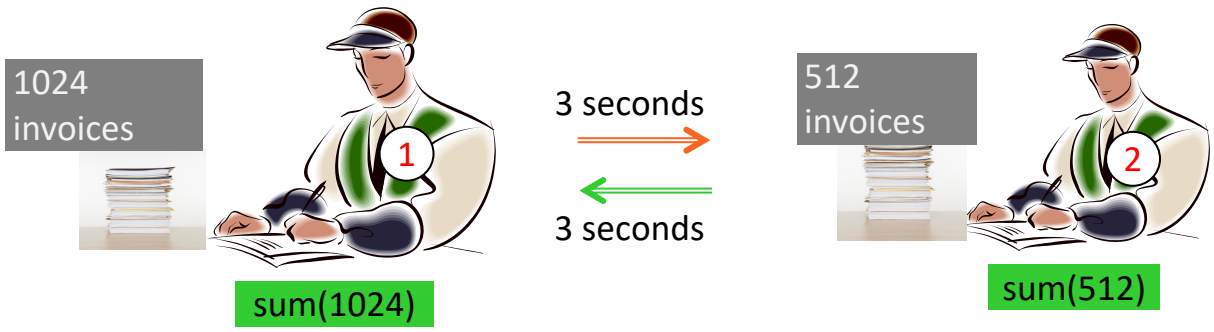
The following assumptions can be used:

1. A stack of 1024 invoices, is initially given to accountant #1. The total amount of all the invoices are to be added.
2. Any accountant takes 1 second to add two numbers. (Counting time = 1 sec)
3. An accountant will require 3 seconds to send any number of invoices or computed results to any other accountant. (Transfer time = 3 sec)
4. Accountant #1 computes the final result.

Case-1: m=1024, number of accountants (n)=1:
T=1023 seconds
only one accountant: this is an example of serial computation.

Distributed and Parallel Computation

Case-2:
m=1024, number of accountants (n)=2 (two accountants)



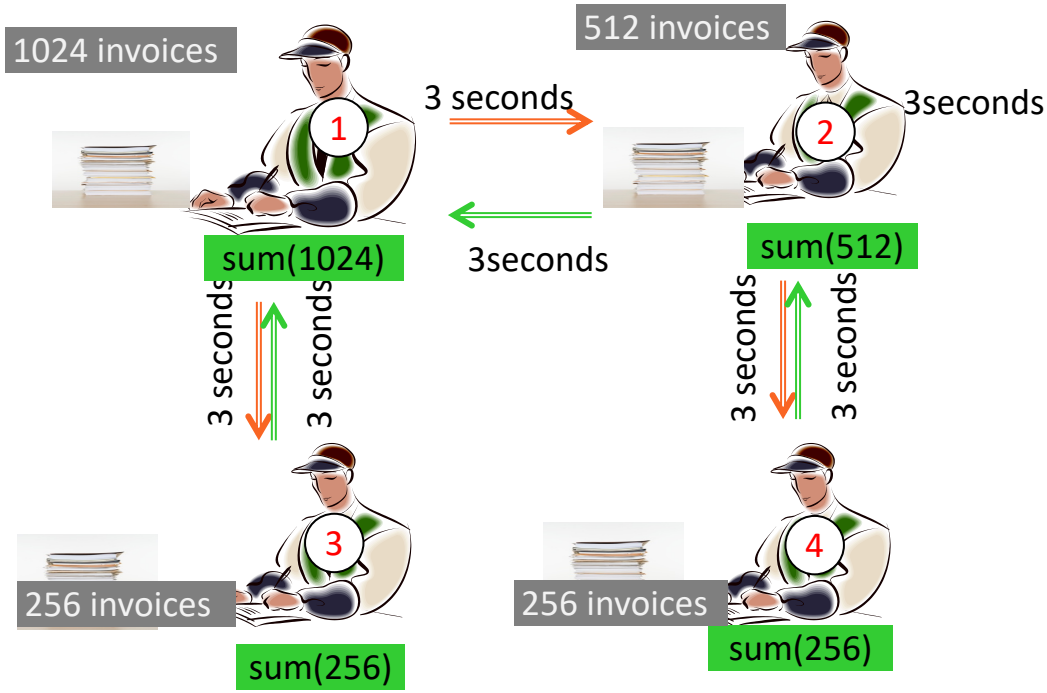
- 3 sec to transfer the invoices to Acc_2
- 511 sec for adding 512 invoices by Acc_1 and Acc_2
- 3 sec to transfer of partial results back to Acc_1
- 1 sec for sum (512) of Acc_1 and Acc_2
- 511×1 seconds to add 512 numbers

- Time to complete the task:

$$T = 3\times1 + 511\times1 + 3\times1 + 1\times1 = 3 + 511 + 3 + 1 = 518\ s$$

DISTRIBUTED AND PARALLEL COMPUTATION

Case-3:
m=1024, number of accountants (n)=4 (four accountants)

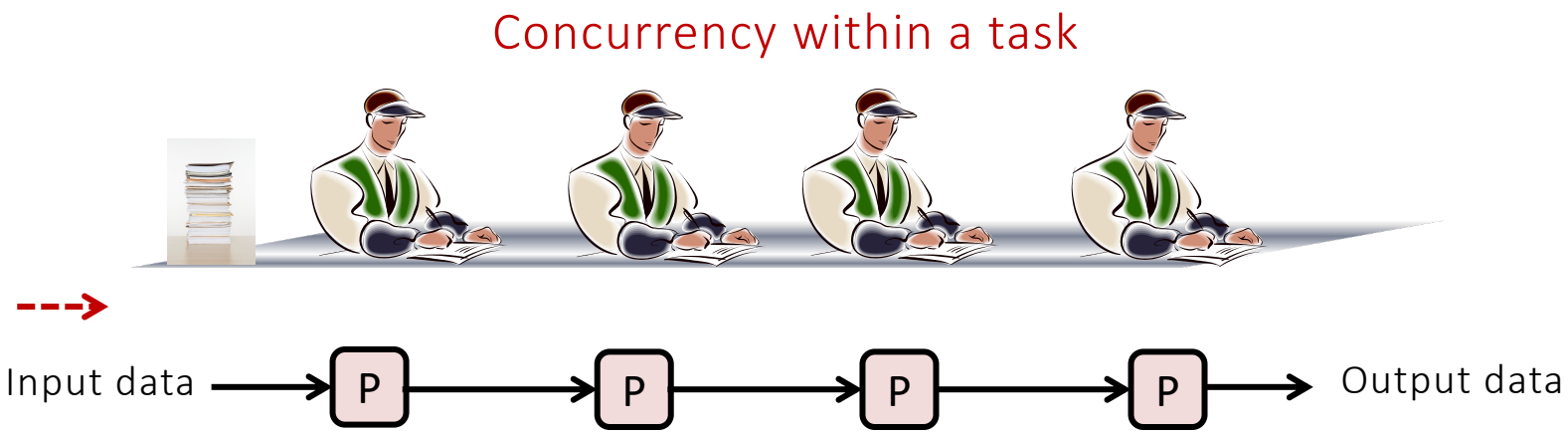


- 3 sec to transfer the invoices to Acc_2
- 3 sec to transfer the invoices to $Acc_3$and Acc_4
- 255 sec for adding 256 invoices by all Accountants
- 3 sec to transfer the partial results back to Acc_1 and Acc_2
- 1 sec for sum(256) by $Acc_1$and Acc_2
- 3 sec to transfer of partial results back to Acc_1
- 1 sec for sum(512) by Acc_1

- Time to complete the task:

- $T = 3\times2 + 255\times1 + 3\times2 + 1\times2 = 6 + 255 + 6 + 2 = 269\ s$

Derived concurrency - Pipelining



- What is the advantage?

  -- Increased throughput.

- Any disadvantage?

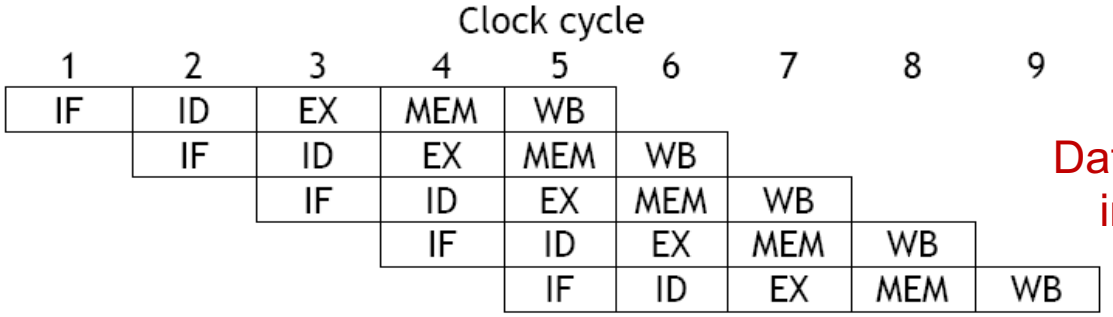
  -- Useful only when there is little/no intra-task dependency.

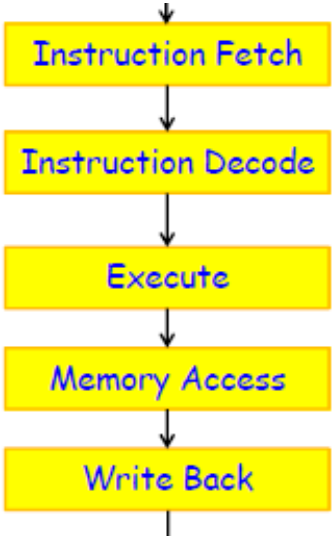
- What about latency?

  -- Increases with the number of pipeline stages.

Derived Concurrency - Pipelining

IF : instruction fetch
ID : instruction decode and data fetch
EX : execute operations or calculate address
MEM : memory operations
WB : write-back to register file





Data access corresponding to previous instruction can be performed while fetching the current instruction (in a same clock cycle)

CPI (cycles per instruction): the value for a pipelined processor is the sum of the base CPI and all contributions from stalls

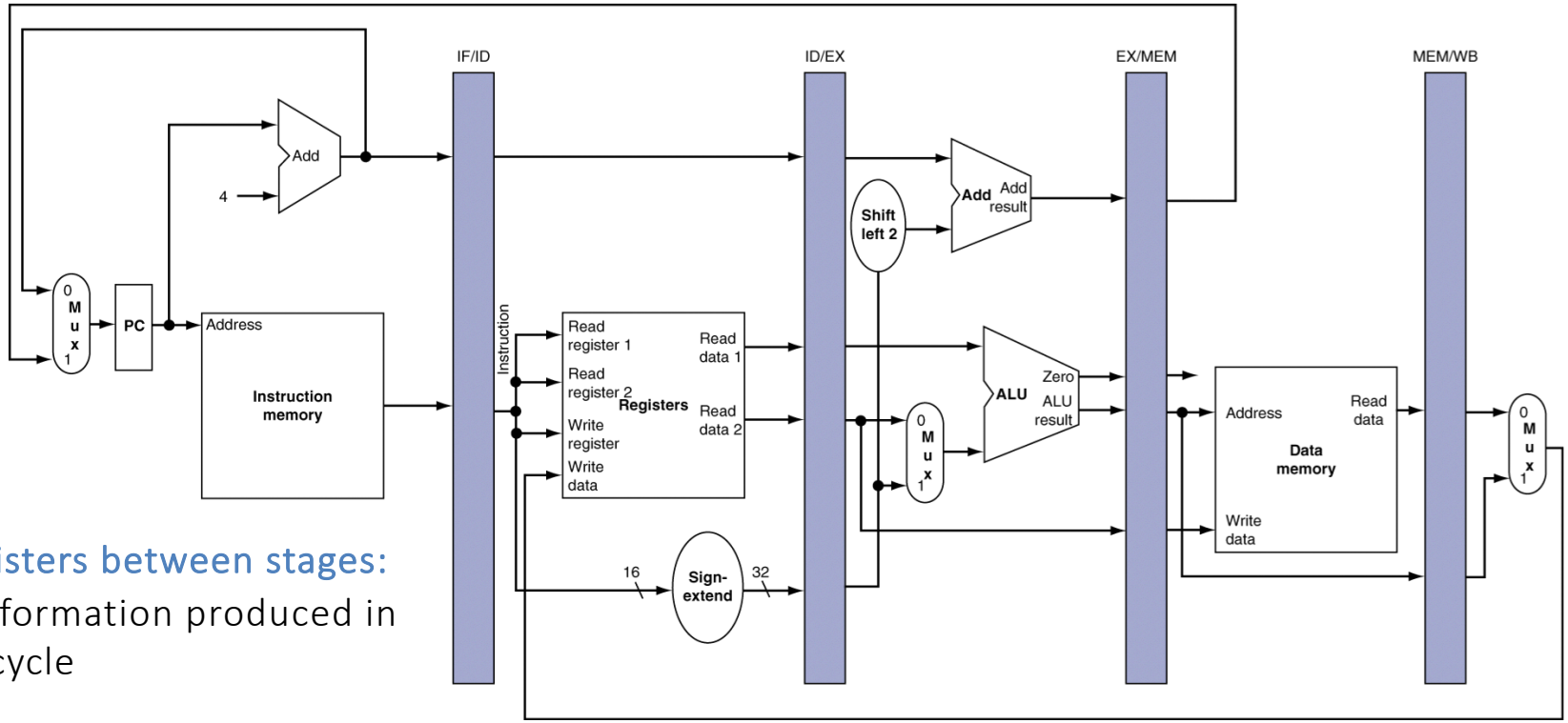
$$\boldsymbol{Pipeline\ CPI}$$

$$= Ideal\ pipeline\ CPI + Structural\ stalls + Data\ harzard\ stalls + Control\ stalls$$

Pipelining Analogy

- MIPS Pipelined
- Five stages, one step per stage:

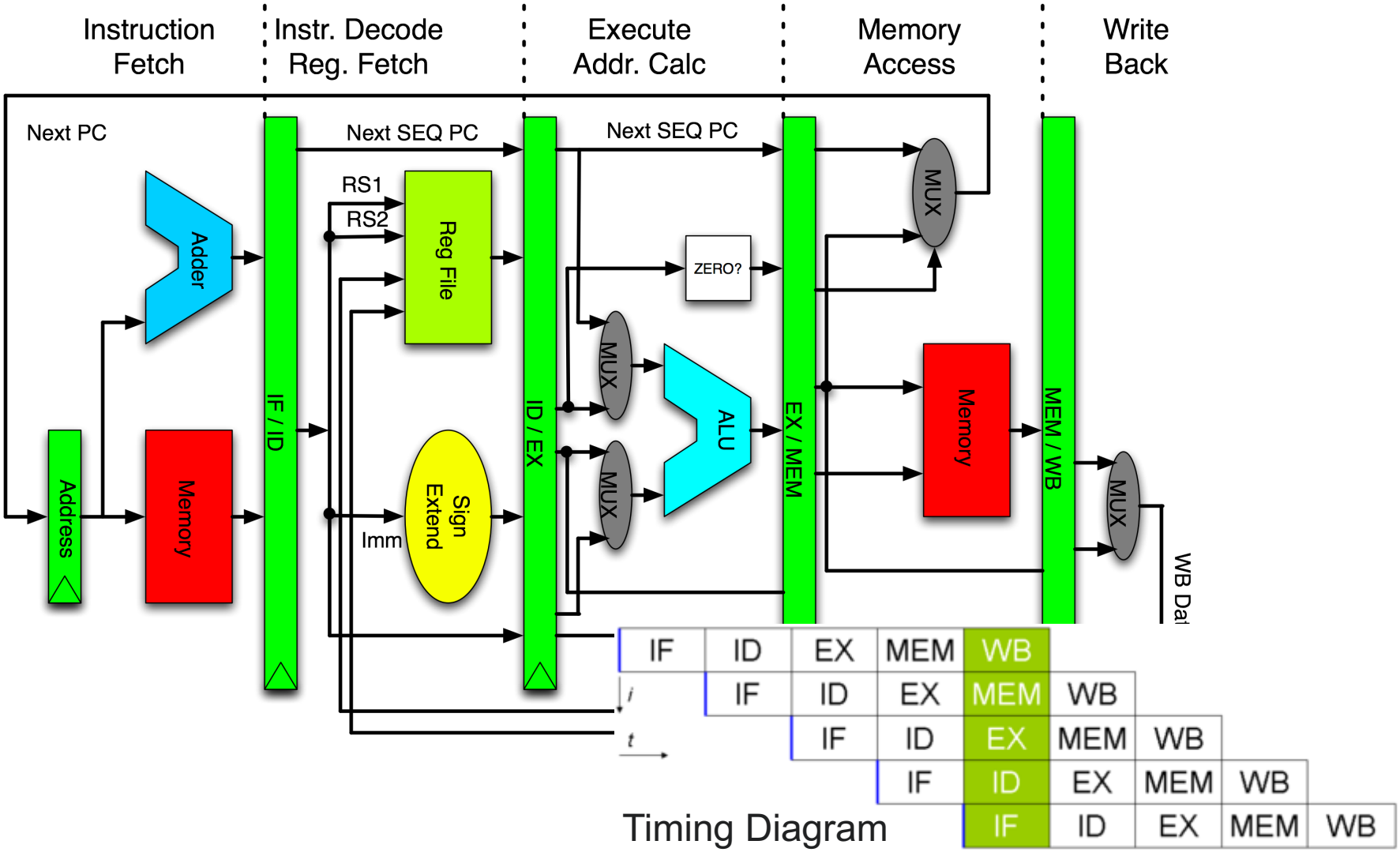
1. IF: Instruction fetch from memory
2. ID: Instruction decode & register read
3. EX: Execute operation or calculate address
4. MEM: Access memory operand
5. WB: Write result back to register



Need Registers between stages:

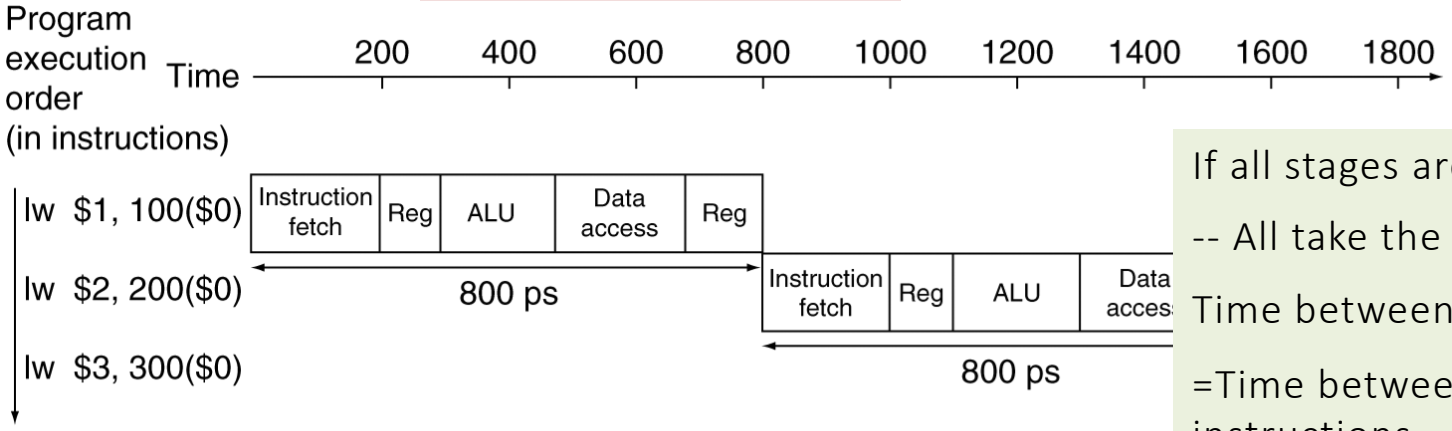
To hold information produced in previous cycle

PIPELINED PROCESSOR

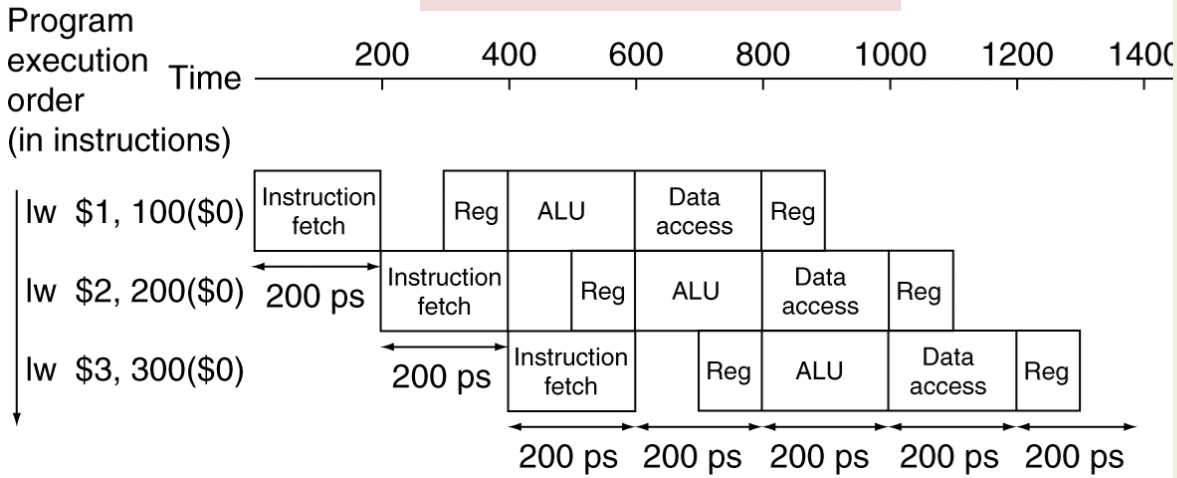


Pipelining Performance

Single-cycle (Tc = 800ps)



Pipelined (Tc = 200ps)



If all stages are balanced:

-- All take the same time

Time between instructions$_{Pipelined}$

=Time between instructions$_{NonPipelined}$/# stages

If not balanced, speedup is less

Speedup is due to the increased Throughput:

Latency (time for each instruction) does not increase

Instruction Level Parallelism (ILP)

- Goals:
  - What is instruction-level parallelism (ILP)?
  - What do processors do to extract ILP?
  - How do processors conduct ILP?

- Why do processors get faster?
  1. More parallelism (or more work per pipeline stage): Get Wider
     -- Fewer clocks/instruction [more instructions/cycle]
  2. Deeper pipelines: Get Deeper
     -- Fewer gates/clock
  3. Transistors get faster (Moore's Law): Get Faster
     -- Fewer ps/gate

Instruction Level Parallelism

- Parallel execution of instructions requires 3 major tasks

1. Which instructions should be executed in parallel?

   -- Checking the data dependence between instructions to identify the instructions which can be grouped together for parallel execution.

2. Where to be executed?

   -- Assigning instructions to different functional units in processor.

3. When to be executed?

   -- Determining when instruction execution is to be initiated.

Instruction Level Parallelism (ILP)

- Extracting yet more performance

1. Increase the depth of the pipeline to increase the clock rate – Superpipelining
   - How does this help performance?
2. Fetch (and execute) more than one instruction at one time (expand every pipeline stage to accommodate multiple instructions) – Multiple-Issue
   - How does this help performance? (impact in performance equation)
   - Today's topic! $\frac{Second}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Second}{Cycle}$
3. Launching multiple instructions per stage allows instruction execution rate, CPI, to be less than 1

- Instead we use IPC (instructions per clock cycle): e.g., a 3 GHz, 4-way multiple-issue processor can execute at a peak rate of 12 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4

Sequential Execution	Instruction-Level Parallelism
1. a = 10 + 5 2. b = 12 + 7 3. c = a + b Instructions: 3 Cycles: 3	1.A. a = 10 + 5 1.B. b = 12 + 7 2. c = a + b Instructions: 3 Cycles: 2 (-33%)

Instruction Level Parallelism

- Two approaches to achieve ILP

1. Hardware Approach: Superscalar Processor

   --P5 Pentium, the first superscalar X86 processor

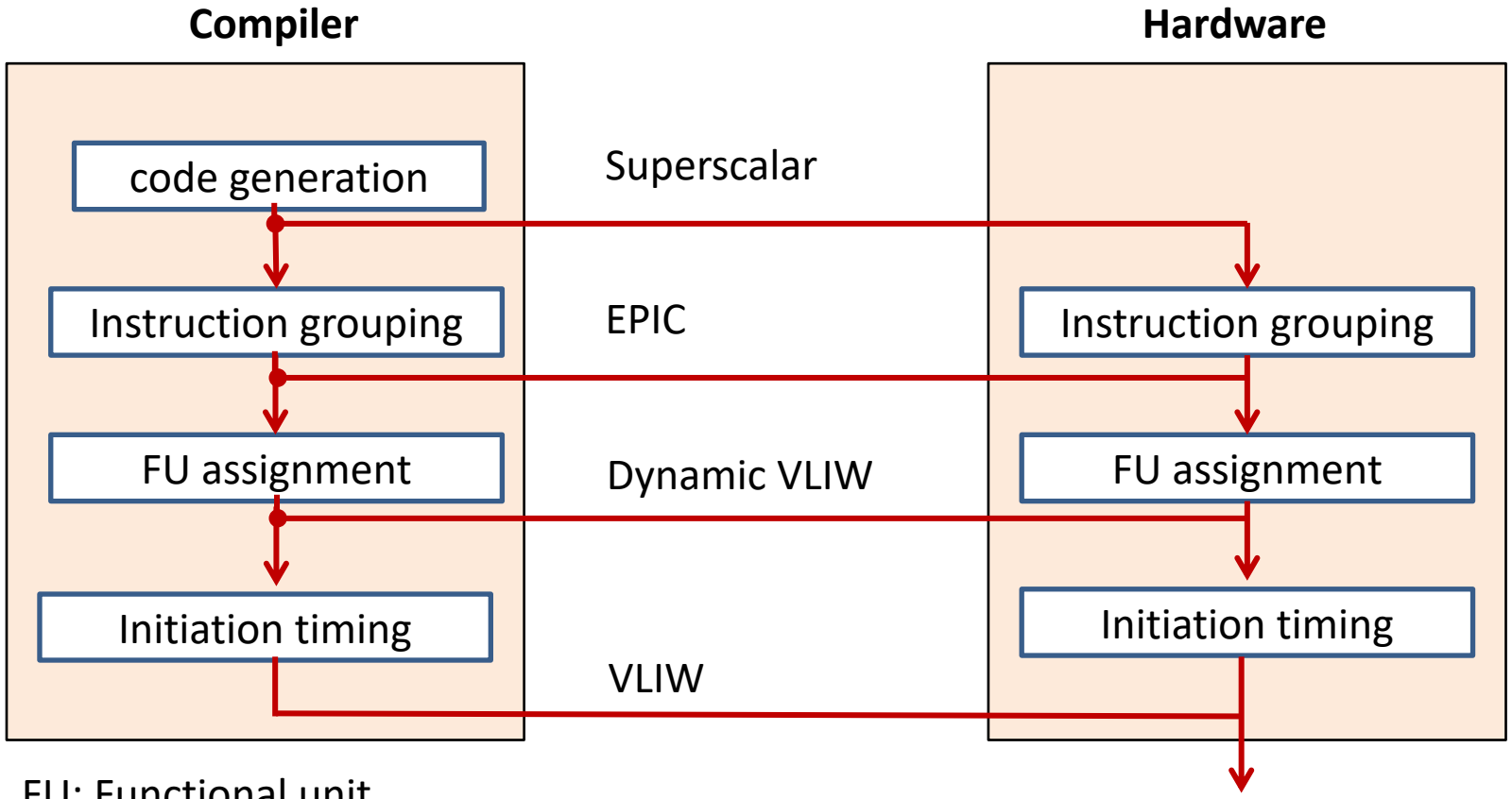
   -- Most general-purpose CPU's developed since 1998 are superscalar

2. Software (or Compiler-based) Approach: Very Long Instruction Word (VLIW) processor

   -- VLIW CPU's contain multiple RISC like Functional Units(FUs). Typically have 4 to 8 FUs.

Instruction Level Parallelism

- Hardware and Software Approaches



FU: Functional unit
EPIC: Explicitly parallel instruction computing

Instruction Level Parallelism

- Methods to extract more parallelism
  - Instruction reordering and out-of-order execution
  - Speculative execution with dynamic scheduling
  - Loop unrolling

-- **Instruction Reordering**: Change the order of execution of instruction if it does not violate the data dependence.

-- **Speculative Execution**: To execute an instruction without exactly knowing if that need to be executed: ahead of branch outcome.

-- **Dynamic Scheduling**: Execute instructions as soon as dependencies are satisfied and functional units are available.

Pipeline Hazards

Review: Pipeline Hazards

Hazards are bad: reduce the amount of achievable machine parallelism and keep us from achieving all the Instruction-Level Parallelism (ILP) in instruction stream

1. Structural hazards

- What? – Resource conflict when HW cannot support all possible combination of instructions simultaneously/overlapped instructions
- How to eliminate? – ***Duplicating resources***

2. Data hazards

- What? – An instruction depends on result of a previous instruction in a way that is exposed by overlapping of instruction in pipeline; need to wait for previous instruction to complete read/write
- How do we eliminate? – ***Forwarding***

3. Control hazards – beq, bne, j, jr, jal

- What? – Pipeline of branches and other instructions that change PC (Program Counter); deciding on control action depends on previous instruction
- How do we eliminate? – ***Insert pipeline bubble***

Review: Pipeline Hazards

- How to deal with hazards?
  - Often, pipeline must be stalled. **Pipeline Stall? Bubble?**
  - Stalling pipeline usually lets some instructions in pipeline proceed, another/others wait for data, resource, etc.

structural hazards

Instr	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3					IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

Data hazards

Instr	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	bubble	EX	MEM	WB			
Instr i+2			IF	bubble	ID	EX	MEM	WB		
Instr i+3				bubble	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

Review: Pipeline Hazards

- Performance of Pipelines with stalls
  - Stalls impede the progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
  - Pipelining can be viewed to decrease CPI or clock cycle time for instruction

$$CPI_{pipelined} = Ideal\ CPI + Pipeline\ stall\ cycles\ per\ instruction$$

$$= 1 + Pipeline\ stall\ cycles\ per\ instruction$$

$$Speedup = \frac{CPI_{unpipelined}}{1 + Pipelining\ stall\ cycles\ per\ instruction}$$

Review: Pipeline Hazards

- Performance of Pipelines with stalls
  - If the unpipelined CPI is equal to the depth of the pipeline:

$$Speedup = \frac{Pipeline\ Depth}{1 + Pipelining\ stall\ cycles\ per\ instruction}$$

$$Speedup from\ Pipeline = \frac{CPI\ unpipelined}{CPI\ pipelined} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

$$= \frac{1}{1 + pipeline\ stall\ cycles\ per\ instruction} \times \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

  - Since:

$$Clock\ cycle\ pipelined = \frac{Clock\ cycle\ unpipelined}{Pipeline\ depth} ==> Pipeline\ depth = \frac{Clock\ cycle\ unpipelined}{Clock\ cycle\ pipelined}$$

  - Therefore, this will lead to the following:

$$\mathbf{Speedup from\ Pipeline == \frac{1}{1 + pipeline\ stall\ cycles\ per\ instruction} \times Pipeline\ depth}$$

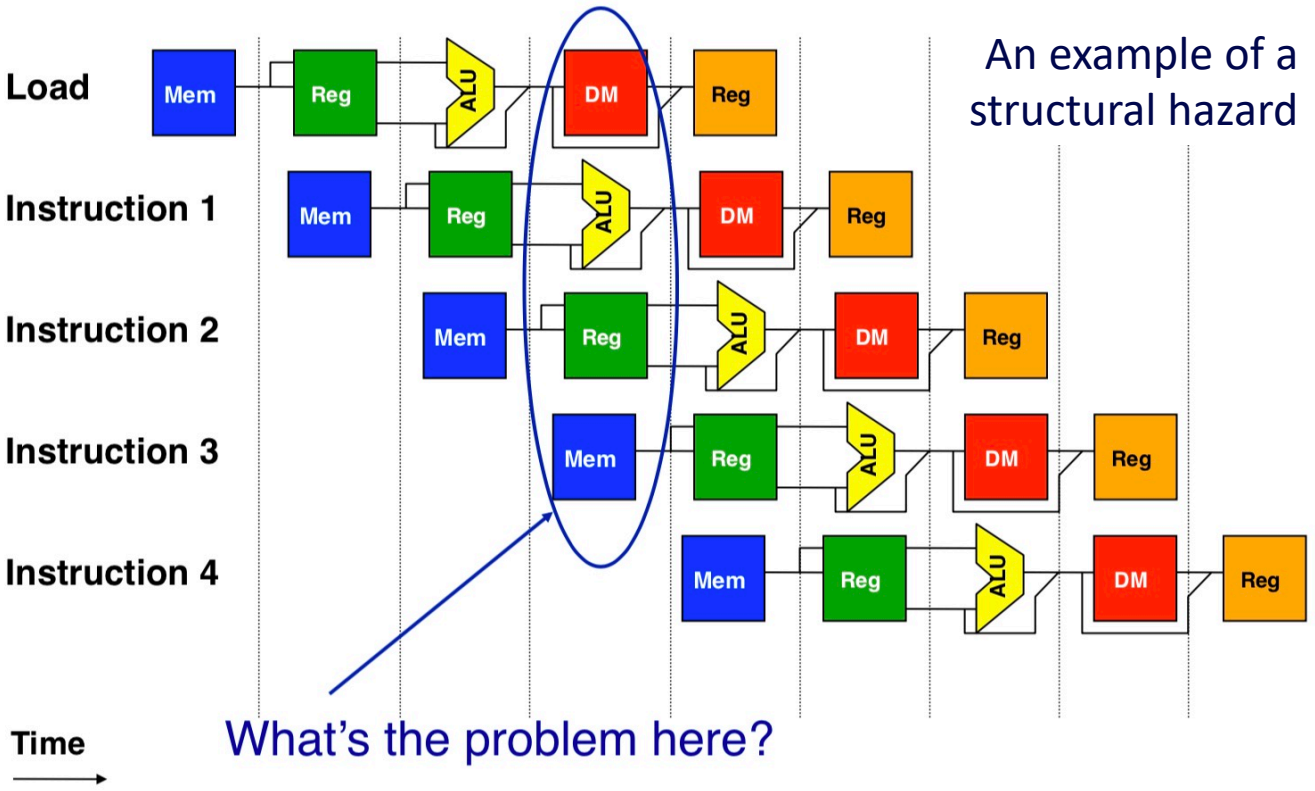
I. Structural hazards

PIPELINE HAZARDS - STRUCTURAL HAZARDS

- If not all possible combinations of instructions can be executed, structural hazards occur
- Avoid Structural hazards by duplicating resources

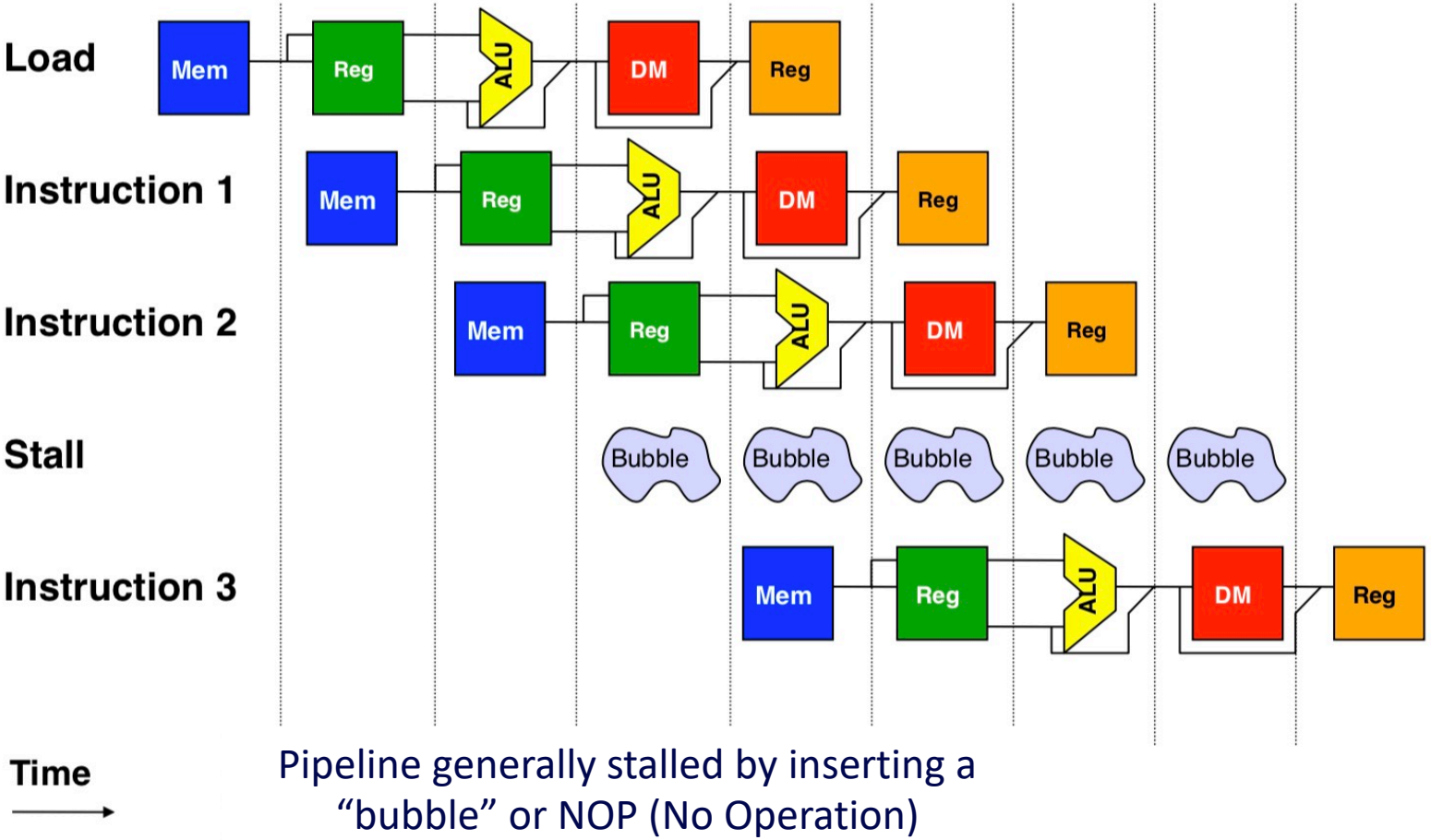
  -- E.g., an ALU to perform an arithmetic operation and an adder to increment PC

- Pipelines stall result of the hazards, the CPI increased from usual "1"



PIPELINE HAZARDS - STRUCTURAL HAZARDS

- How is it resolved?



Pipeline generally stalled by inserting a "bubble" or NOP (No Operation)

PIPELINE HAZARDS - STRUCTURAL HAZARDS

- Or alternatively…

Clock Number →

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. i+1		IF	ID	EX	MEM	WB				
Inst. i+2			IF	ID	EX	MEM	WB			
Inst. i+3				stall	IF	ID	EX	MEM	WB	
Inst. i+4						IF	ID	EX	MEM	WB
Inst. i+5							IF	ID	EX	MEM
Inst. i+6								IF	ID	EX

- LOAD instruction "steals" an instruction fetch cycle which will cause pipeline to stall
- Thus, no instruction completes on clock cycle 8

- What is the realistic solution?

  -- Answer: Add more hardware

Especially for the memory access example, i.e., common case, CPI degrades quickly from our ideal "1" for even the simplest of cases

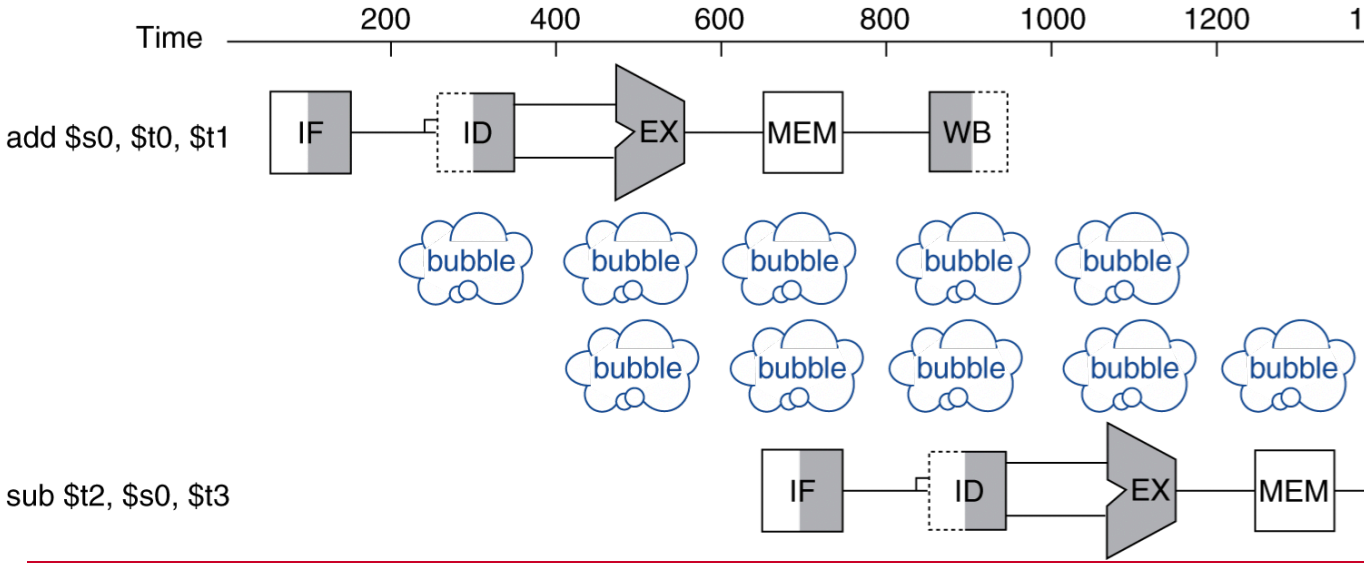
II. Data hazards

PIPELINE HAZARDS - DATA HAZARDS

Data hazards exit because of pipelining

- Why do they exist?
  - An instruction depends on completion of data access by a previous instruction; Pipelining changes when data operands are read, written
  - Order differs from order seen by sequentially executing instructions on un-pipelined machine

```
-- ADD $s0, $t0, $t1
-- sub $t2, $s0, $t3
```



- All instructions after ADD use the result of ADD
- ADD writes the register in WB but SUB needs it in ID

This is the data hazards

Pipeline Hazards - Data hazards

- Data hazards specifics
- There are actually 3 different kinds of data hazards
  - Read After Write (RAW)
    $Instr_J$ tries to read operand before $Instr_I$ writes it

    ```
    I: add r1,r2,r3
    J: sub r4,r1,r3
    ```

    → Results from an actual need for communication

    – Caused by a "**dependence**" (in compiler nomenclature).
  - Write After Read (WAR)
    $Instr_J$ writes operand *before* $Instr_I$ reads it

    ```
    I: sub r4,r1,r3
    J: add r1,r2,r3
    ```

    – Called an "**anti-dependence**" by compiler writers.
  - Write After Write (WAW)
    $Instr_J$ writes operand *before* $Instr_I$ writes it.

    ```
    I: sub r1,r4,r3
    J: add r1,r2,r3
    ```

    – Called an "**output dependence**" by compiler writers

- Result from the reuse of name "r1",
- Can't happen in MIPS 5 stage pipeline

PIPELINE HAZARDS - DATA HAZARDS

Data hazards and the compiler

- Compiler should be able to help eliminate some stalls caused by data hazards

-- i.e., compiler could not generate a LOAD instruction that is immediately followed by instruction that uses result of LOAD's destination register

What about the control logic?

- For MIPS integer pipeline, all data hazards can be checked during ID phase of pipeline
- If data hazard, instruction stalled before its issued
- Whether forwarding is needed can be determined at this stage, control signals set
- If hazard detected, control unit of pipeline must stall pipeline and prevent instructions in IF, ID from advancing

Situation	Example	Action

No Dependence	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	LW R1, 45(R2)
	ADD R5, R1, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX

Dependence overcome by forwarding	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R1, R7
	OR R9, R6, R7
		Comparators detect the use of R1 in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX

Dependence with accesses in order	LW R1, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R1, R7
		No action is required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Overcome Data hazards: Dynamic Scheduling

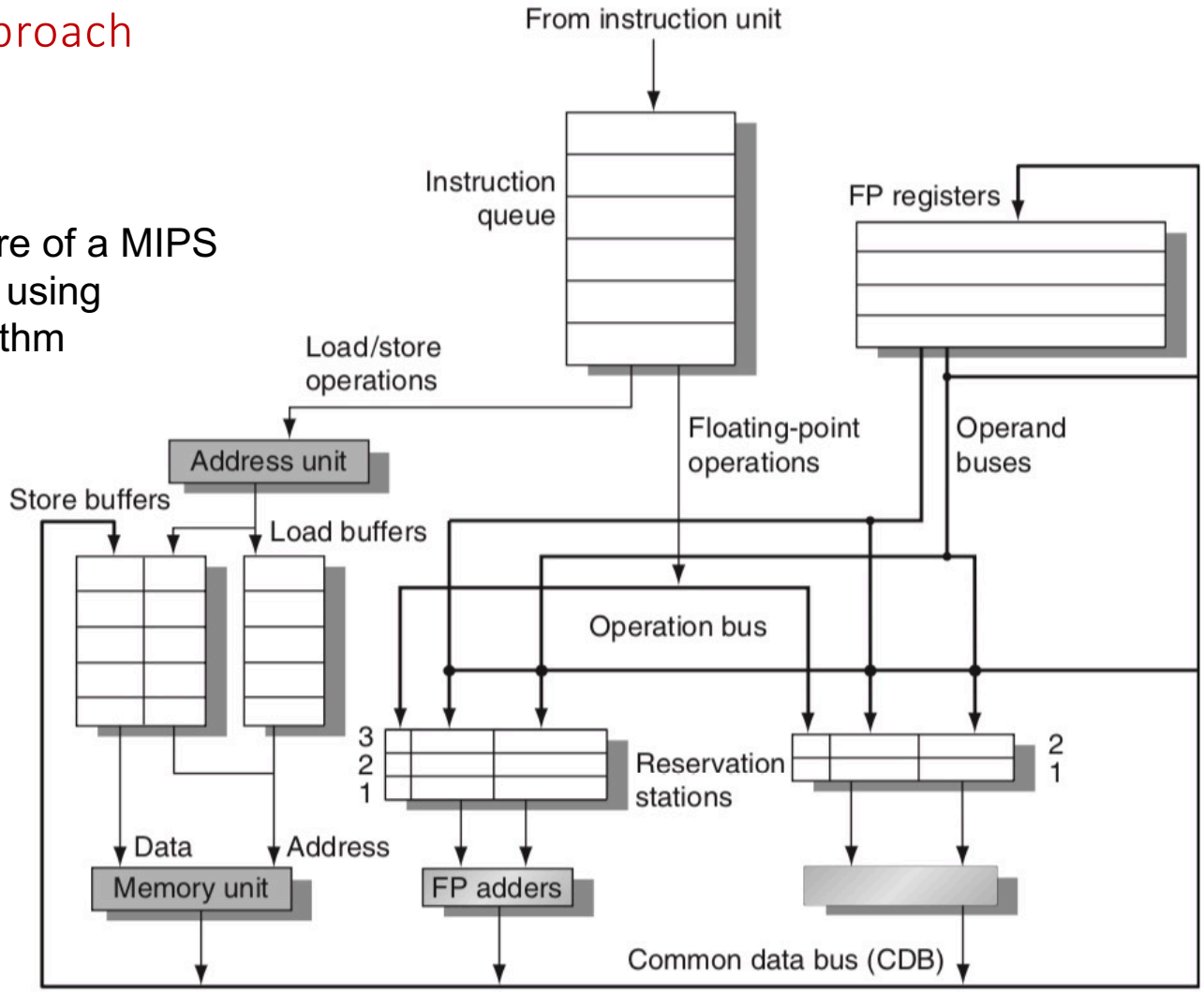
- Dynamic scheduling
  - It allows code that was compiled with one pipeline to run on another different pipeline
  - It can handle some cases when dependences are unknown at the compile time
  - It allows the processor to tolerate unpredictable delays

- Dynamic Scheduling Using Tomasulo's Approach

Overcome Data hazards: Dynamic Scheduling

- Tomasulo's Approach

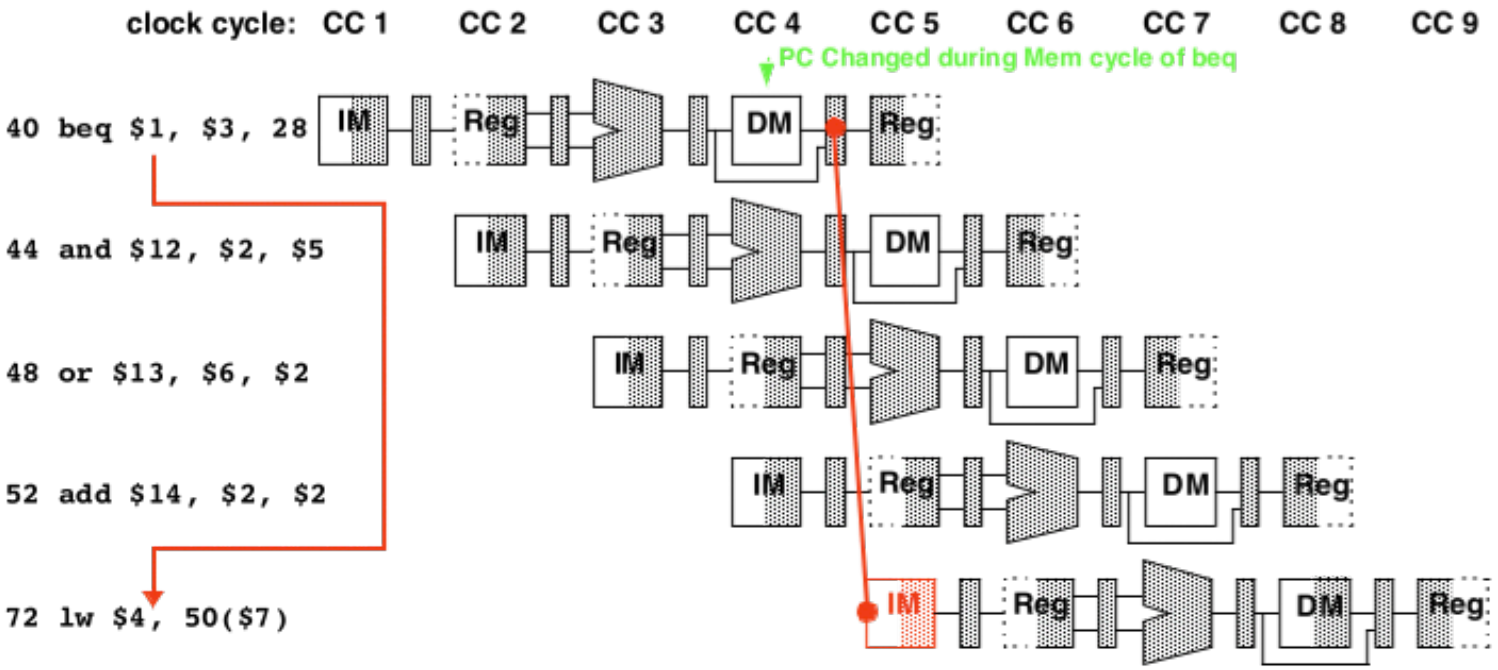
The basic structure of a MIPS floating-point unit using Tomasulo's algorithm



III. Control hazards (Branch Hazard)

PIPELINE HAZARDS - CONTROL HAZARDS

- How branches impact pipelined instructions



- If branch condition is true, must skip 44, 48, 52
  - -- But, these have been already started down the pipeline
  - -- They will complete unless we do something about it

Pipeline Hazards - Control hazards

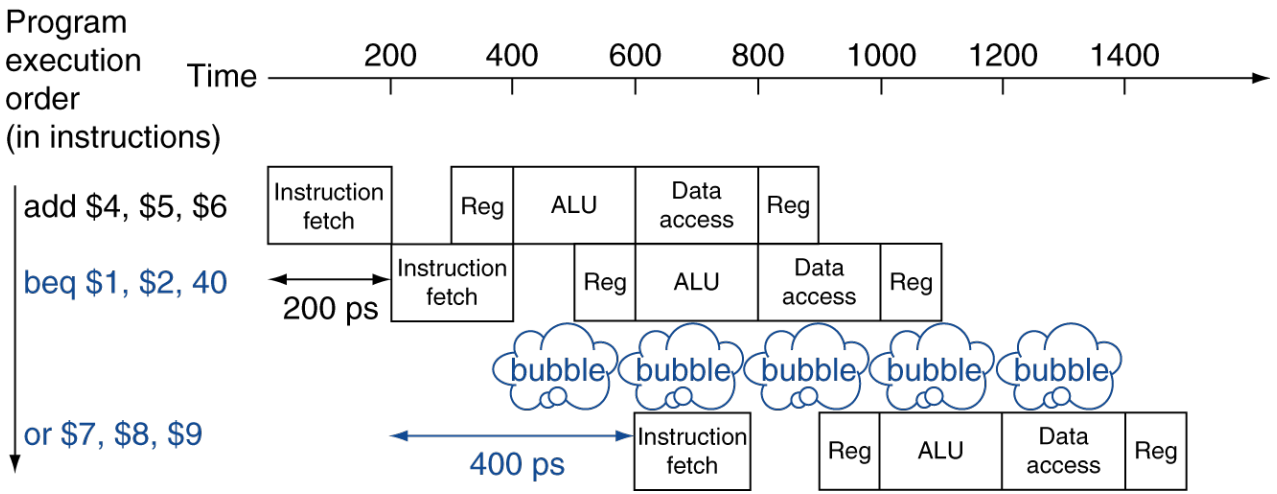
- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline cannot always fetch correct instructions

    --Still working on ID stage of branch

- Reduce Pipeline Branch Penalties
  - Freeze or flush pipeline
  - Predicted-not-taken (Predicted-untaken)
  - Treat every branch as taken
  - Delayed Branch

PIPELINE HAZARDS - CONTROL HAZARDS

- Dealing branch hazards: always **stall**
  - Wait until branch outcome is determined before fetching next instruction



  - If CPI = 1, 25% branch:

    -- Stall 1 cycle => new CPI = 1.25

    -- Stall 2 cycle => new CPI = 1.5

    -- Stall 3 cycle => new CPI = 1.75

Pipeline Hazards - Control hazards

- Dealing branch hazards: always **stall**

- On average, branches are taken ½ the time
  -- If branch is not taken
    * Continue normal processing
  -- Else, if branch is taken..
    * Need to flush improper instruction from pipeline

- One approach:
  -- Always assume branch will NOT be taken
    * Cut overall time for branch processing in ½
  -- If prediction is incorrect, just flush the pipeline

Pipeline Hazards - Control hazards

- Dealing branch hazards: Branch Prediction (Static & Dynamic)

- Better (& more common): guess possible outcome
  1. Technique is called "branch predicting"; needs 2 parts:
     * "Predictor" to guess where/if instruction will branch; and to "where"
     * "Recovery Mechanism": i.e., a way to fix the mistake
  2. Prior strategy:
     * "Predictor" : always guess branch never taken
     * "Recovery" : flush instructions if branch taken
  3. Alternative: accumulate info. In IF stage as to…
     * "Whether or not" for any particular PC value a branch was taken next
     * To "where" it is taken
     * "How" to update with information from later stages

Pipeline Hazards - Control hazards

- Dealing branch hazards: Branch Prediction (Static & Dynamic)

- Static Branch Prediction
  -- Based on typical branch behavior
  -- Example: loop and if-statement branches
    * Predict backward branches taken
    * Predict forward branches not taken

- Dynamic Branch Prediction
  -- Hardware measures actual of each branch: e.g., record recent history of each branch
  -- Assume future behavior will continue the trend
    * When wrong, stall while re-fetching, and update the history

Pipeline Hazards - Control hazards

Dynamic Branch Prediction

- **Why does prediction work?**

  -- Underlying algorithm has regularities

  -- Data that is being operated on has regularities

  -- Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems

- **Is dynamic branch prediction better than static?**

  -- Seems to be

  -- There are a small number of important branches in programs having dynamic behaviors

- **Two pieces of information needed for a branch**

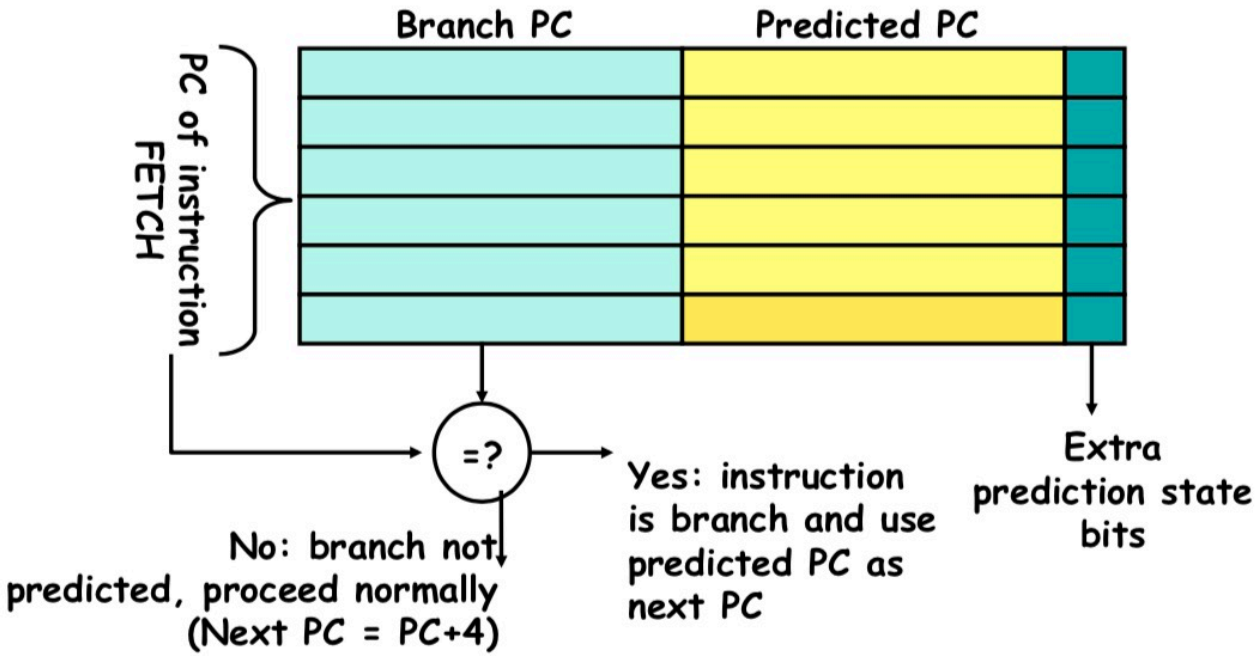
  -- Direction (whether the branch is taken or not), provided by Branch Predictor

  -- Target (If it is taken, where it does), provided by Branch Target Buffer (BTB)

PIPELINE HAZARDS - CONTROL HAZARDS

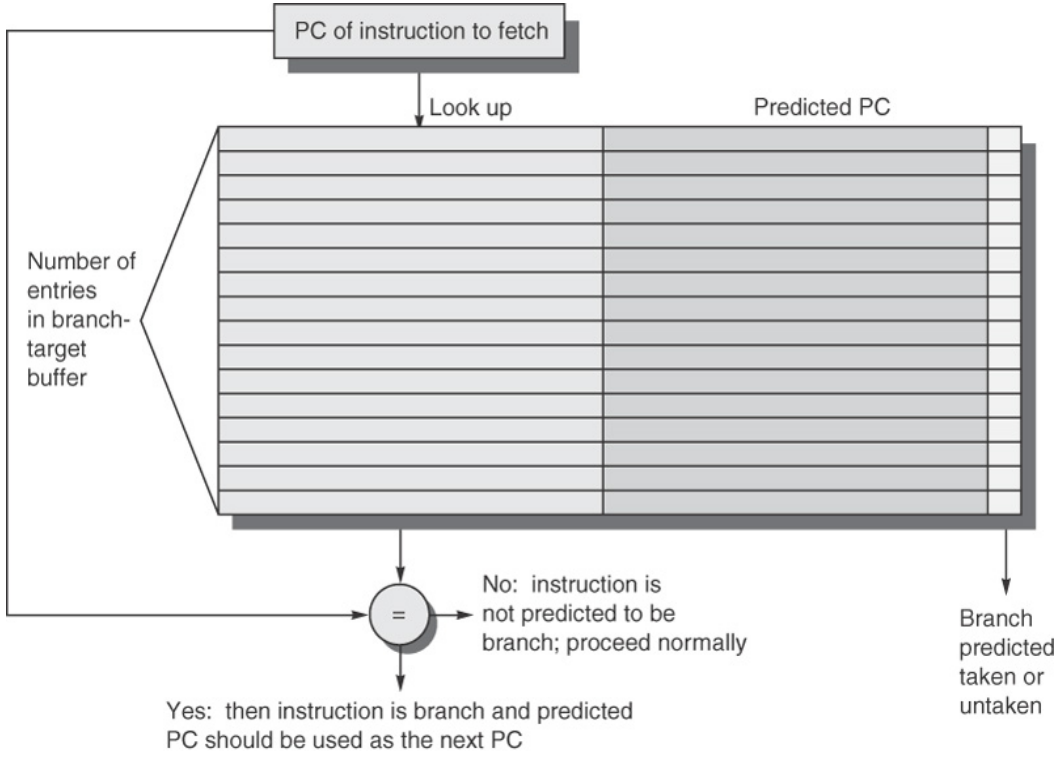
- Branch Target Buffer (BTB)
  - BTB: Address of branch index to get prediction AND branch address (if taken)

    -- Note must check for branch match now, since can't use wrong branch address
  - Example: BTB combined with BHT



Pipeline Hazards - Control hazards

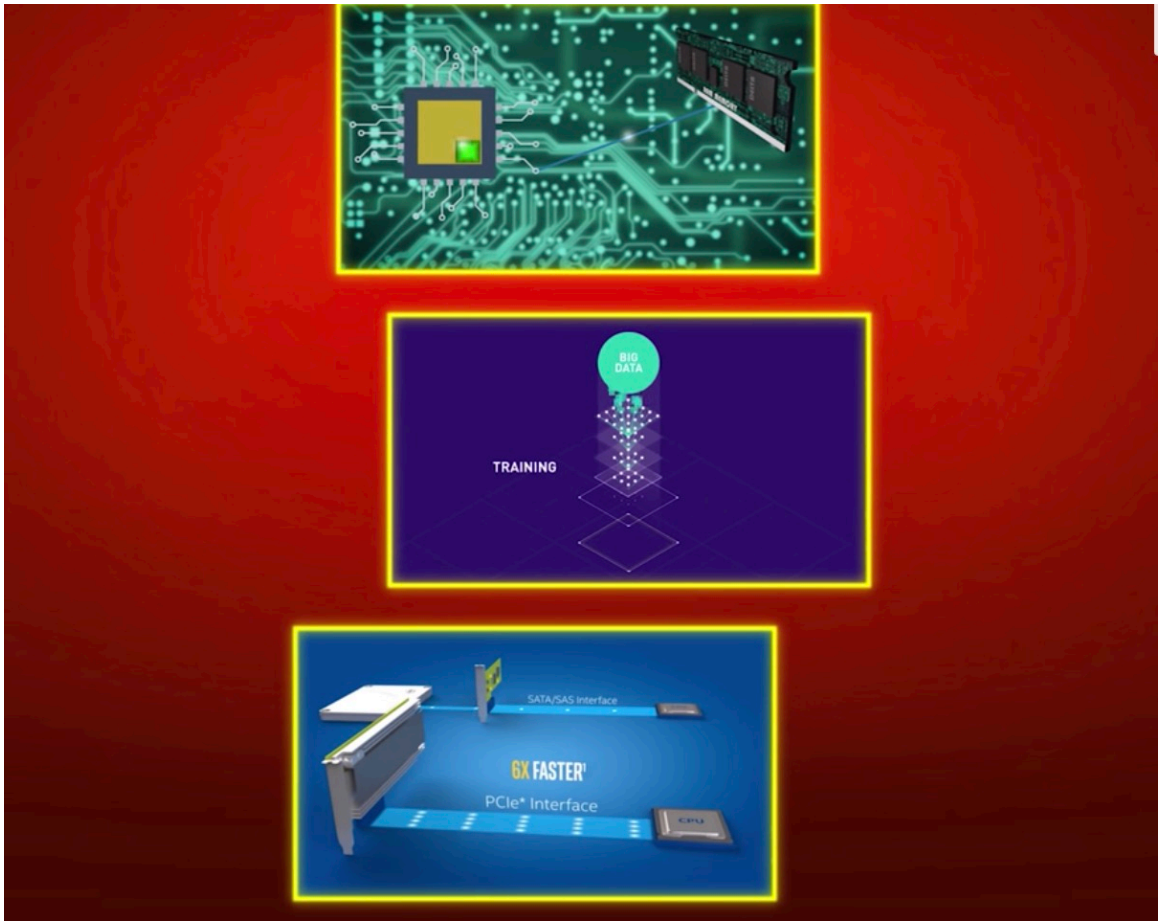
- Calculate the branch target
  - The BPB predicts when a branch is taken, but does not tell where its taken to!
  - Even with predictor, still need to calculate the target address
    - 1-cycle penalty for a taken branch
  - Branch target buffer
    - Cache of target addresses
    - Indexed by PC when instruction fetched (if hit and instruction is branch predicted taken, can fetch target immediately)



Custom and Emerging Computing Trends

The Future of Computing

Heterogeneous Architecture – CPUs, GPUs, FPGAs, ASICs, ...

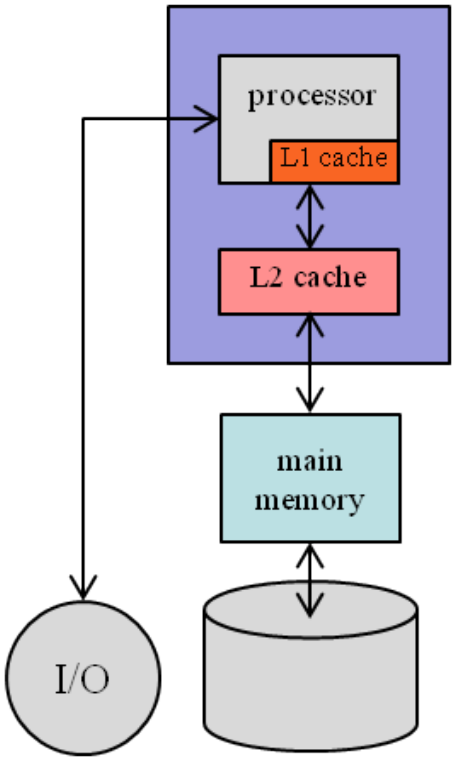


Overview

- Specific architectures:
  - Examples: ASIP, FPGA, and ASIC
  - Comparison of general-purpose processors: DSP, GPU
- Heterogeneous multicore platforms
  - Introduction to domain specific computing

General Purpose Processor (GPP)

- Hardware features:
  - The program to be run and necessary data could be made available at main memory
  - General data-path: consists of a general ALU and usually a large register file
  - Multiple levels of cache for reducing memory latency
- Maximum flexibility
  - Programmable: support several high-level languages
  - Can be used for any general applications
- Other key features…
  - Short time-to-market
  - Low non-recurring engineering (NRE) cost
  - High power consumption

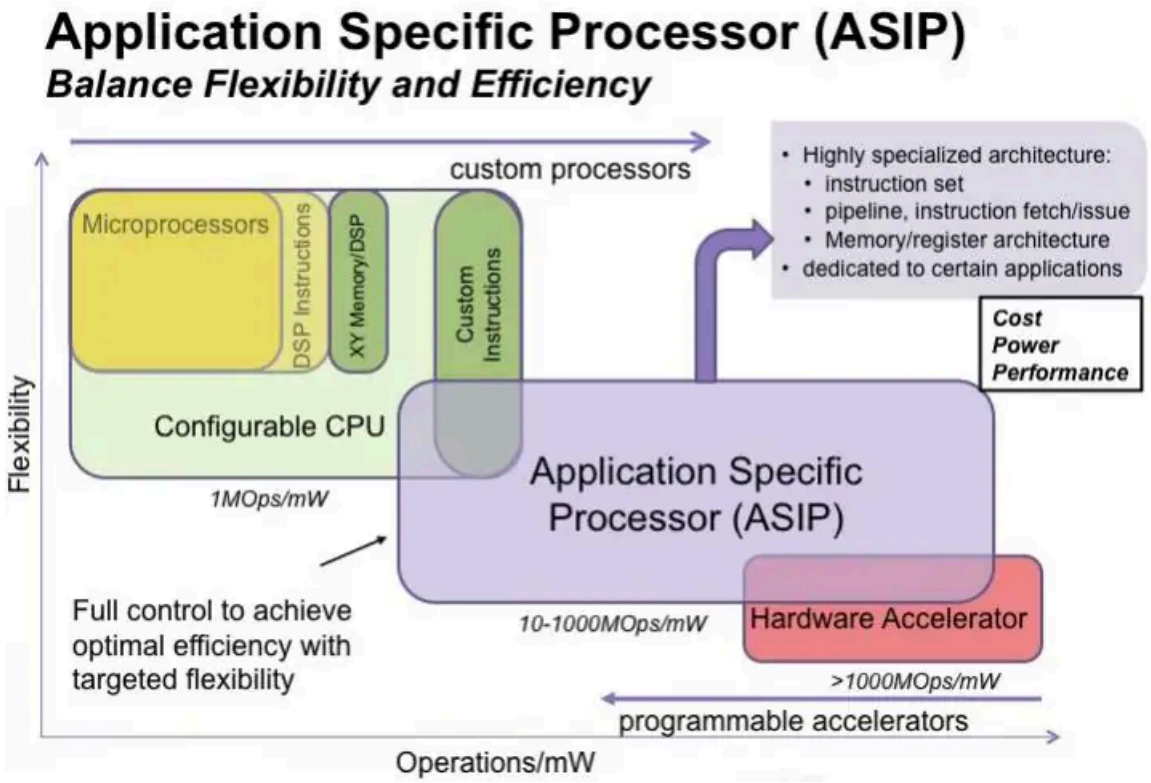


Application Specific Instruction Set Processor (ASIP)

- A microprocessor tailed to benefit a specific application or a domain of applications (operations)

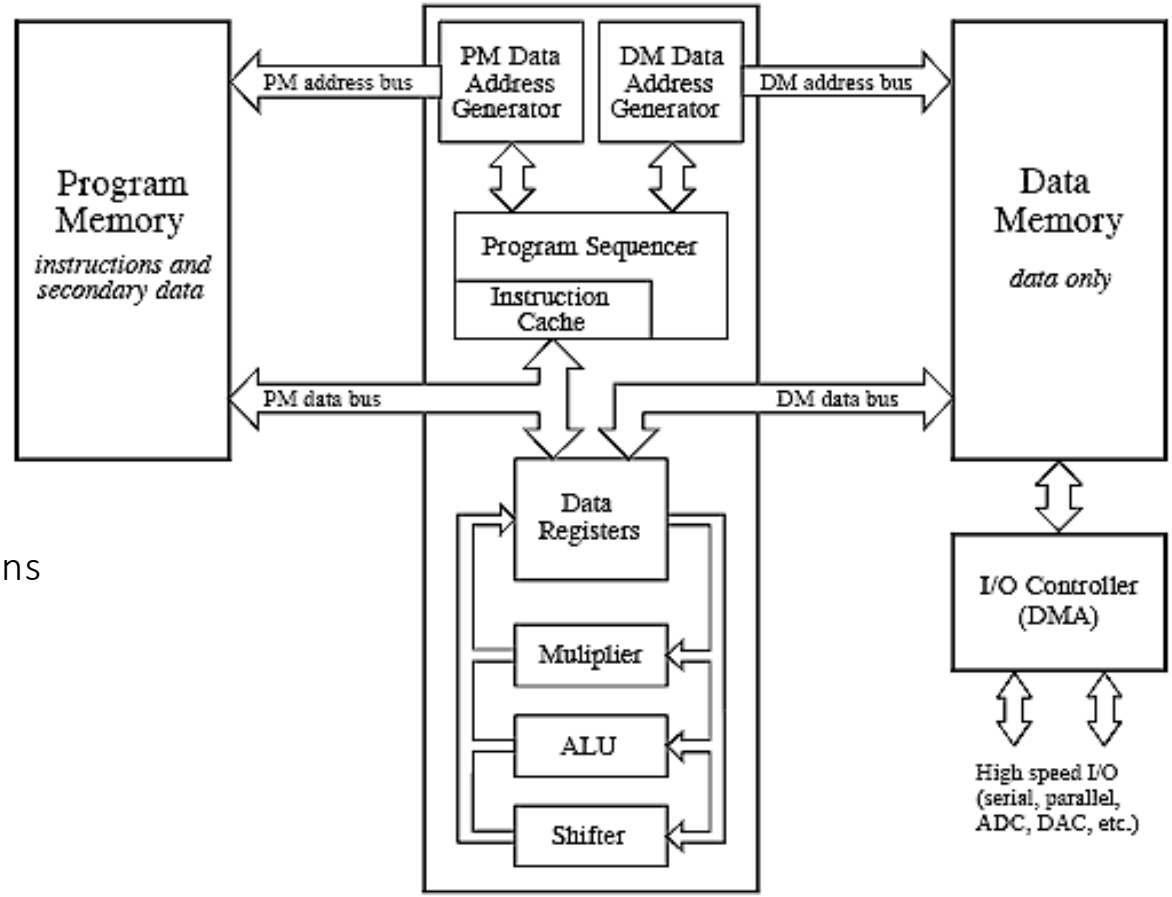
  -- Signal processing, image processing, video processing, digital communication, etc.

- Instruction set of ASIP is customized for the type of computation involved in specific application
- Requirement on flexibility should be sufficient instead of unlimited like GPP
- Highest performance
- Lower power consumption
- Smaller silicon, design cost



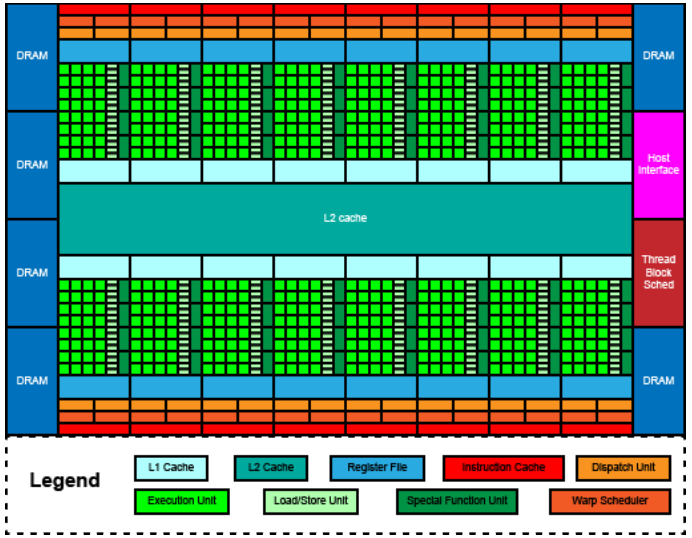
Digital Signal Processor (DSP)

- A Digital Signal Processor (DSP) is an ASIP designed for repetitive multiply accumulate operation and bit-reversal addressing
- Real-time processing of computation-intensive tasks while minimizing:
  - Cost (than GPP)
  - Power consumption
  - Specialized datapath
  - Specialized instruction set for DSP
- Repetitive numeric calculations
- High memory bandwidth
- Support streaming data
- Less development time



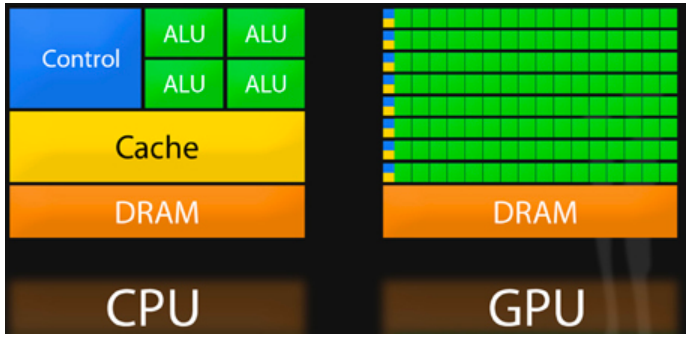
Graphic Processing Unit (GPU)

- GPU is a processor – Specialized for graphics
- Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel
- Heterogeneous execution model (CPU is the host, GPU is the device)



- The motivation of GPGPU computing:

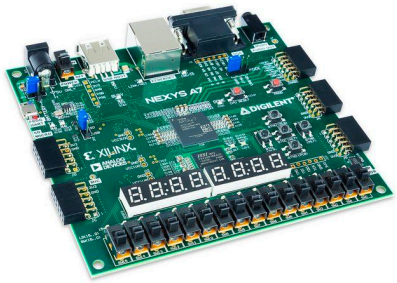
-- Parallel computing by massively data-parallel stream processing
-- Use thousands of threads; Implementation in hundreds of cores
-- Cost effective: cheap
-- Using already available hardware



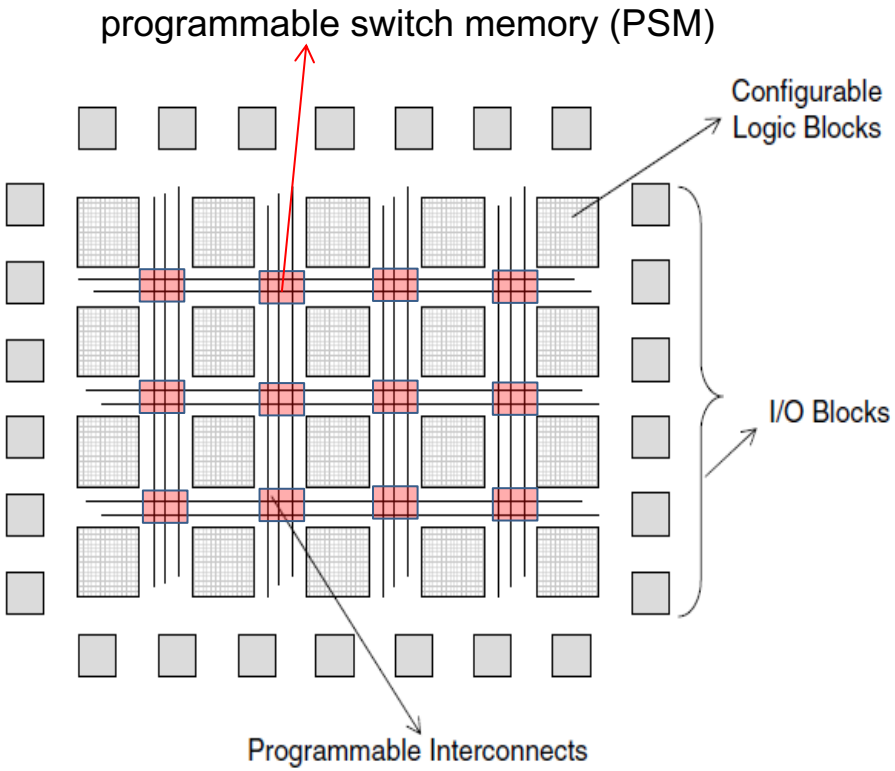
- GPUs involve high latency, high throughput processing
- CPU performs low latency low throughput computation

Field Programmable Gate Array (FPGA)

- Core architecture of FPGA consists of three main components:
  1. An array of configurable/ programmable logic blocks:
     -- combinational units
  2. A sea of programmable interconnects
  3. Memories and specialised I/O blocks



- Other components in modern FPGAs:
  - ✓ DSP blocks
  - ✓ Hardwired IP cores
  - ✓ Soft cores
  - ✓ …



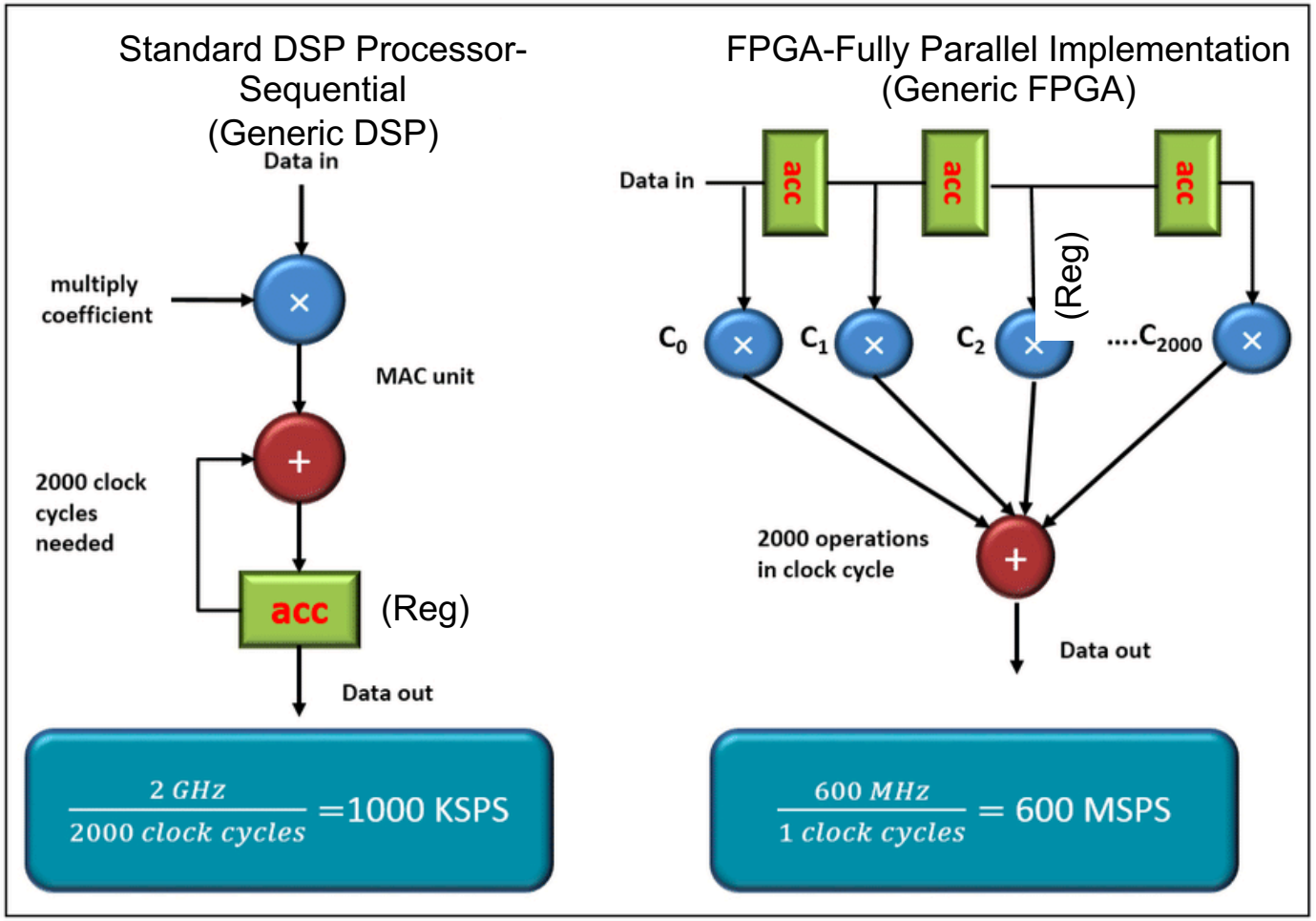
FPGA vs. DSP

1. Both FPGA and DSP offer flexibility for reuse and programmability

2. DSP has an advantage in time-in-market

3. FPGA supports parallel design and offers the greater performance

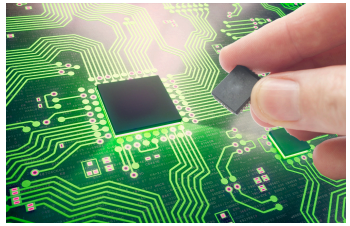
4. FPGA is more expensive and power-consuming

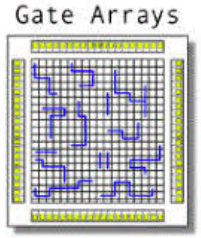


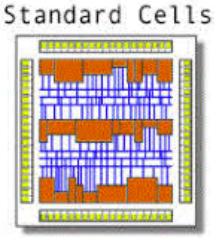
Heterogenous arch. With DSP and FPGA components are an emerging trend

Application Specific Integrated Circuits (ASIC)

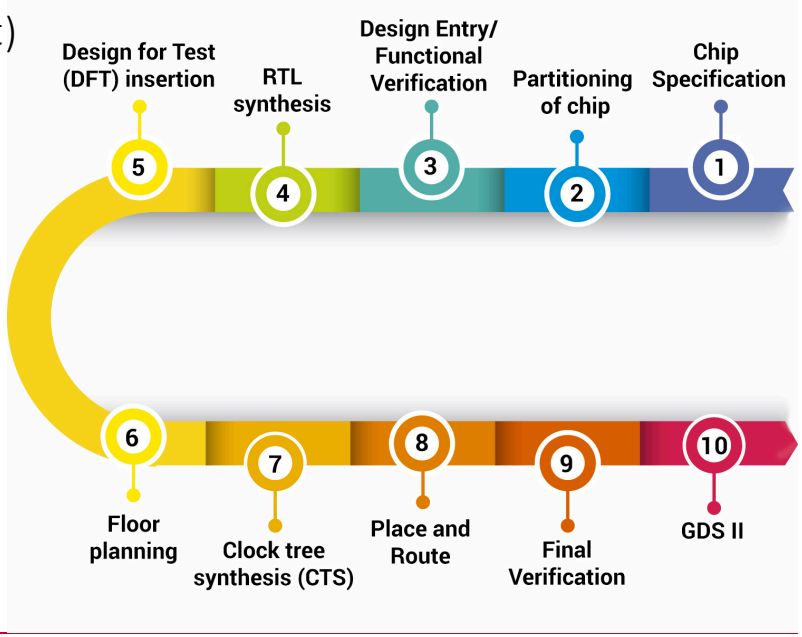
- Provide customized tailed HW solutions for specific apps. or problems



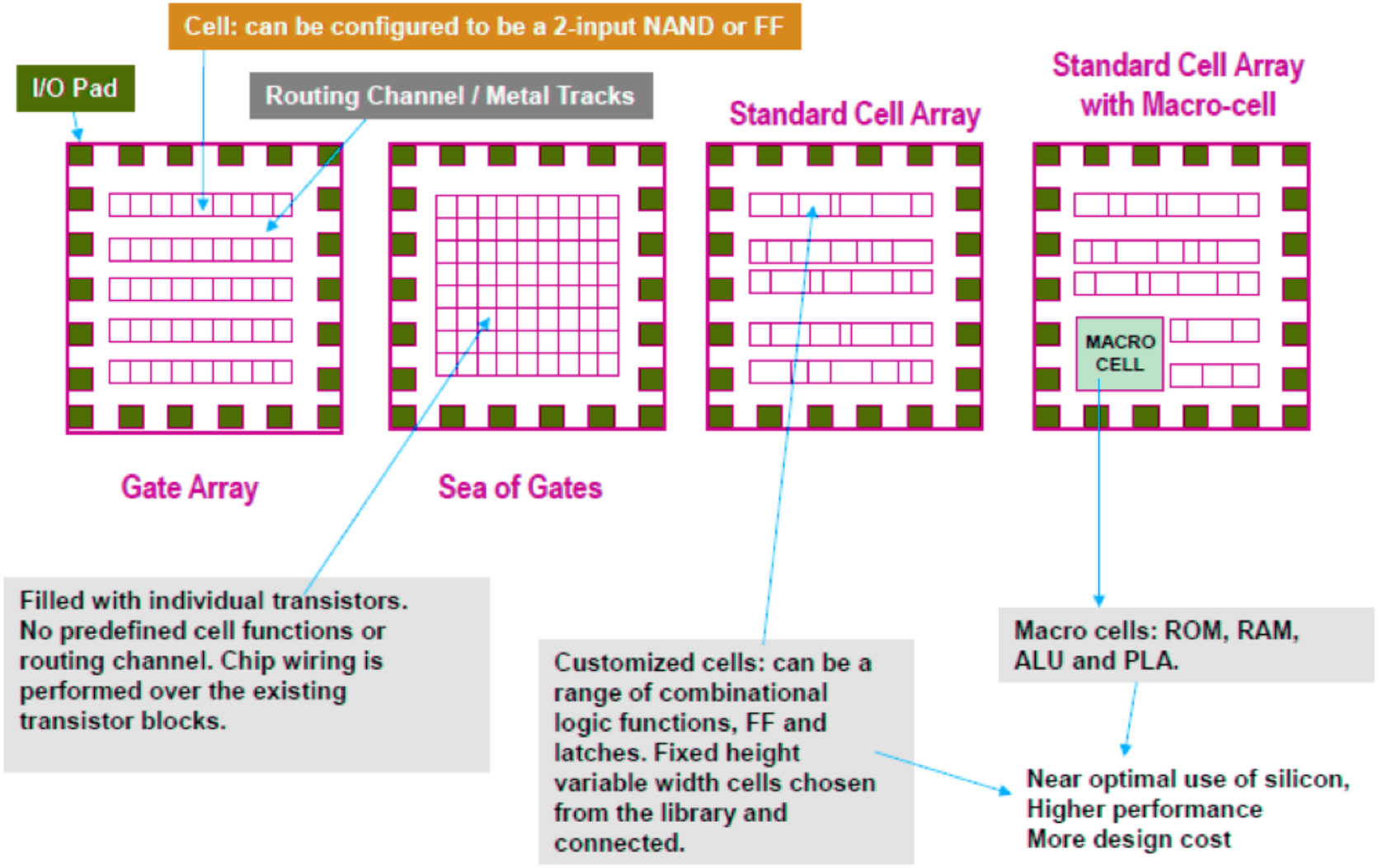




  - Not configurable
  - Cannot be upgraded, updates require redesign
  - Involve high Non-recurring engineering (NRE) cost of several millions for development and testing (need to have large volume market to reduce cost)
  - Longer design/development time (time-to-market)
  - Maximum performance per watt, per unit area
  - Possible design optimizations
  - Semi-custom design: **gate array** implementation, **standard array** implementation
  - Full-custom design: cell design, cell library development and use, simulation and testing for design verification

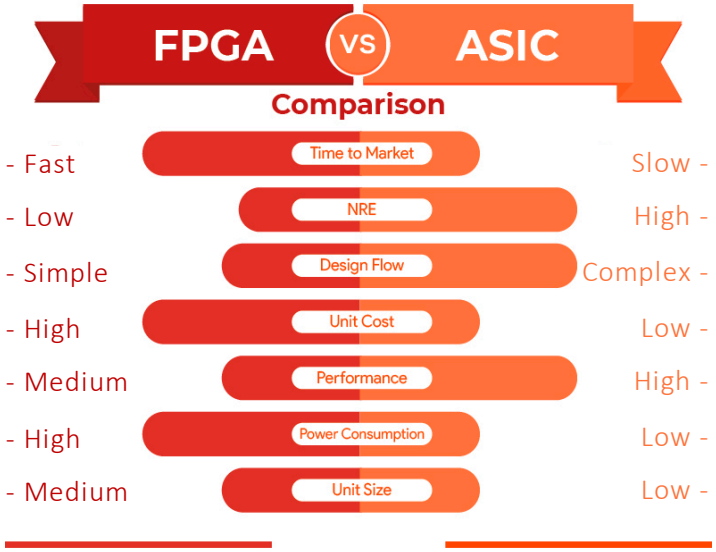


GATE ARRAYS AND STANDARD CELLS



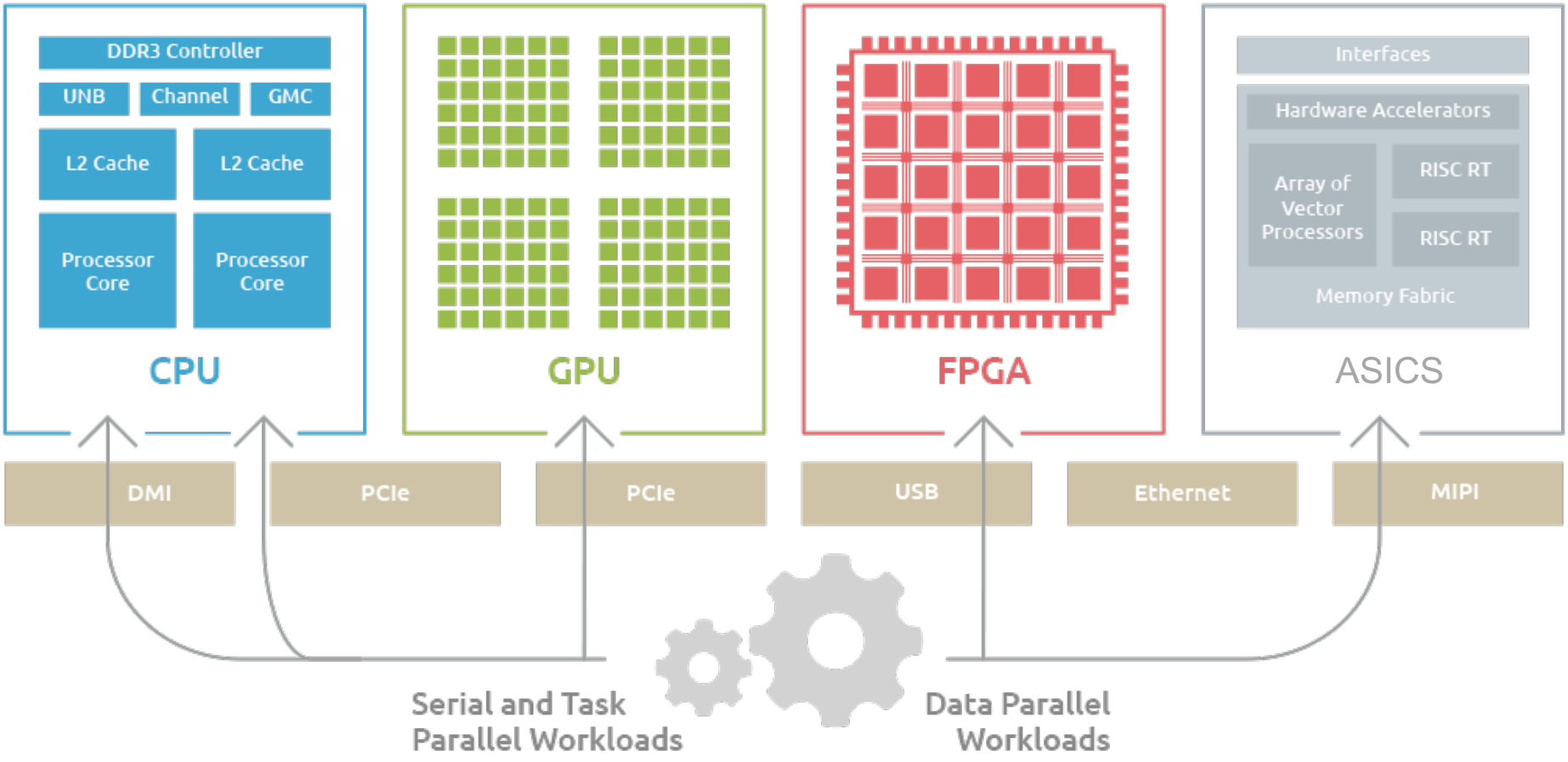
FPGA vs. ASIC

- Differences
- FPGA:
  - ✓ Reconfigurable, shorter development time
  - ❖ more power and energy, expensive for large volume market
- ASIC:
  - ✓ higher clock Freq, higher speed, smaller
  - ❖ Development tools are expensive, not flexible

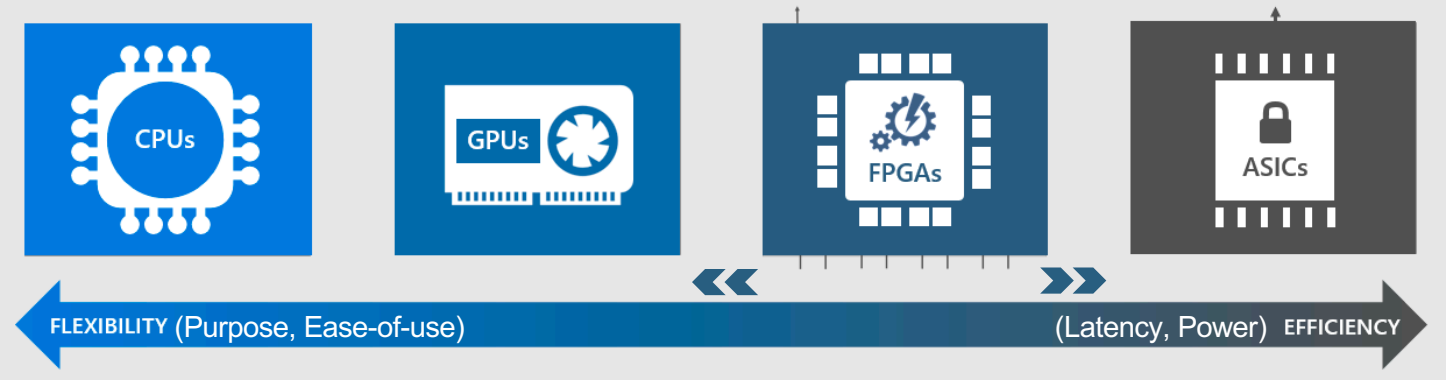


- Integration benefits:
  - ✓ both have highly parallel tasks to provide greater performance,

CPU vs. GPU vs. FPGA vs. ASIC

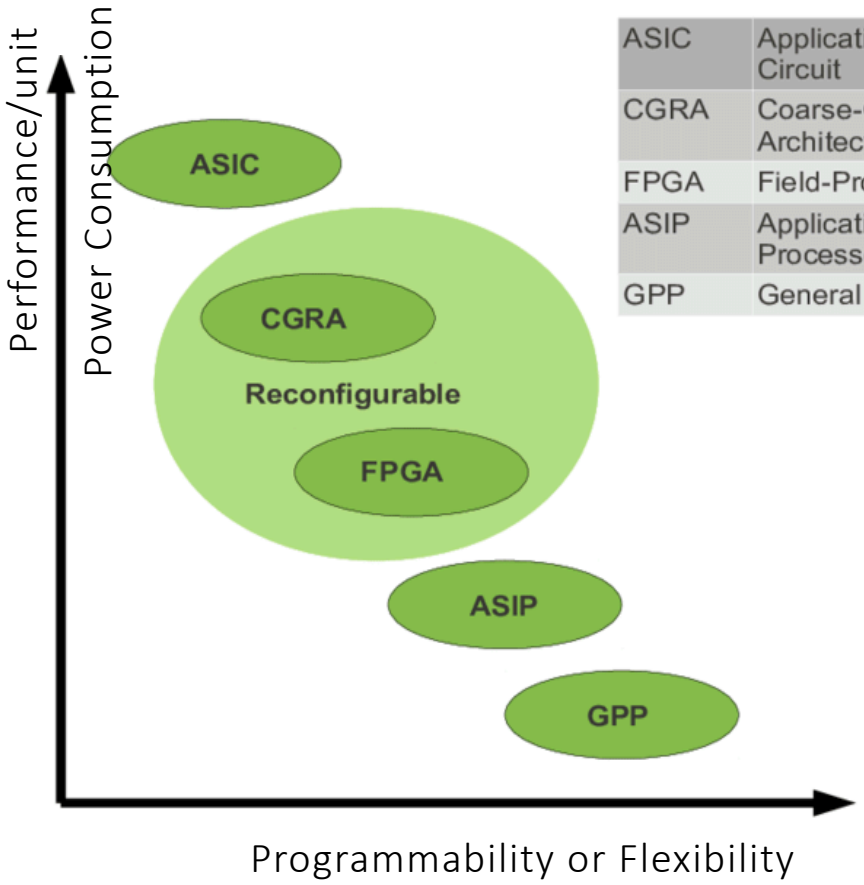


CPU vs. GPU vs. FPGA vs. ASIC

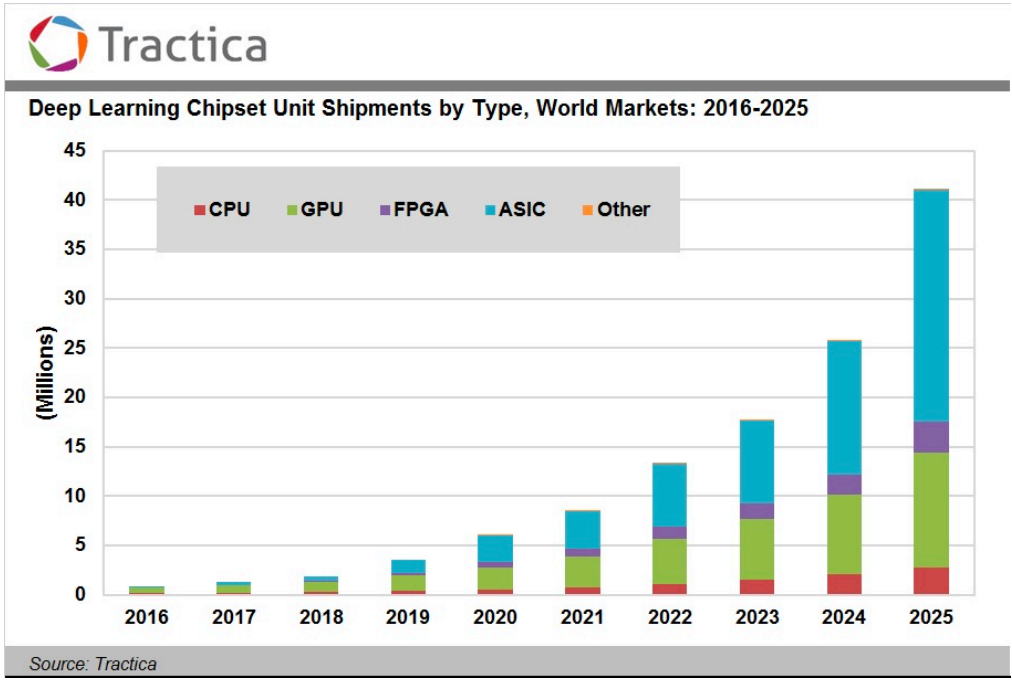


	CPU	GPU	FPGA	ASIC
Compute Adaptability (to a variety of situations)	High	Medium	Low	None
Compute power	Medium	High	High	Medium
Latency	Medium	High	Low	Ultra low
Throughput	Low	High	High	High
Parallelism	Low	High	High	High
Power efficiency	Medium	Low	Medium	High

COMPARISON



ASIC	Application Specific Integrated Circuit
CGRA	Coarse-Grained Reconfigurable Architecture
FPGA	Field-Programmable Gate Array
ASIP	Application-Specific Instruction Set Processor
GPP	General Purpose Processor

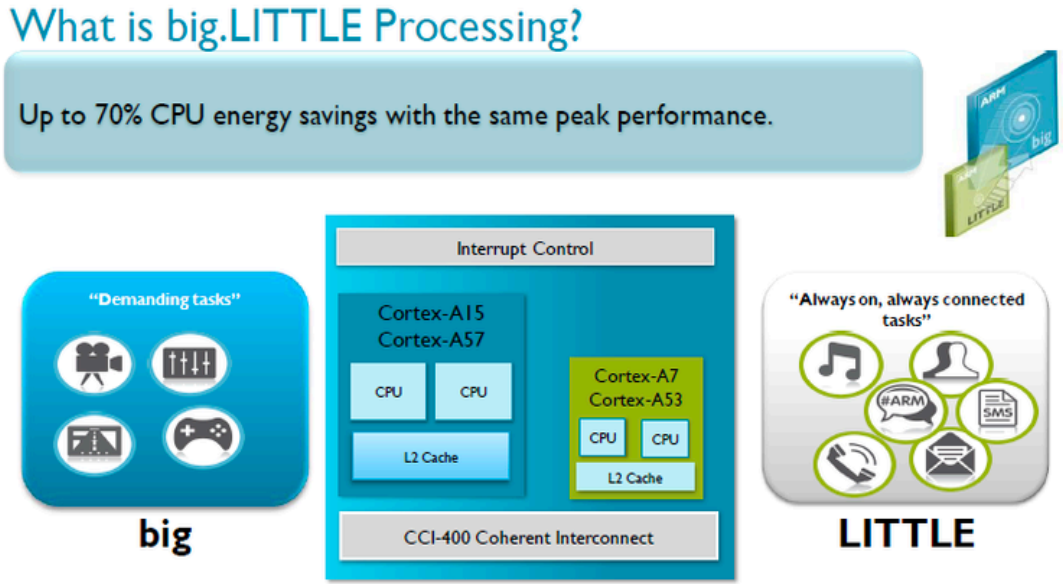


HETEROGENEOUS MULTICORE PLATFORMS

- ARM big. LITTLE

  -- Selects processor according to job to minimise power consumption

  -- Cores need not be always on



Two different types of cores:

- big: ARM Cortex-A15: consists of out-of-order multi-issue pipeline high-performance CPUs

  -- Does the computation-intensive tasks

- LITTLE: ARM Cortex-A7: consists of in-order, 8-stage pipeline CPUs

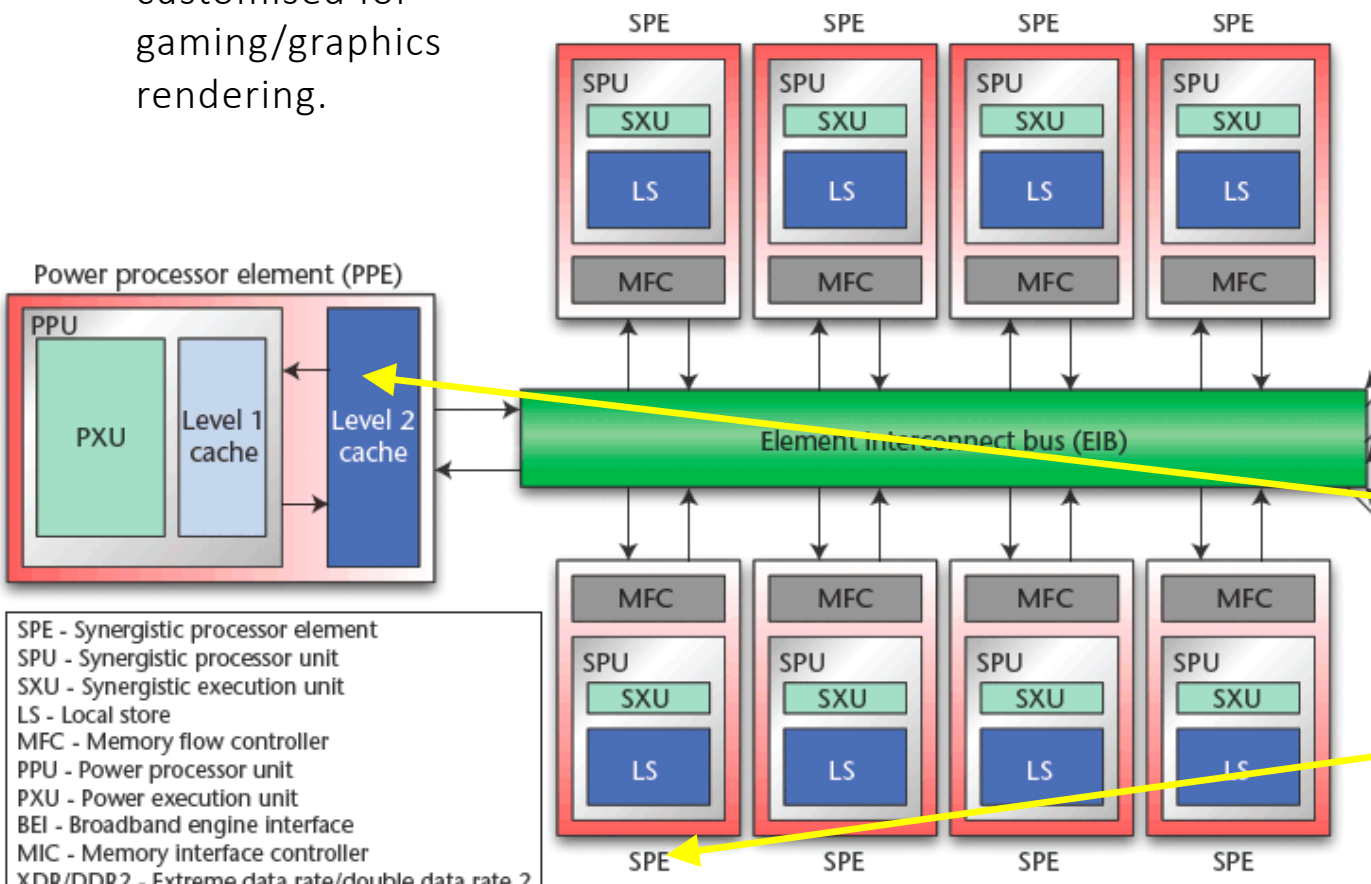
  -- Perform tasks which demand less computation

HETEROGENEOUS MULTICORE PLATFORMS

- IBM Cell Broadband Engine (BE) architecture

- Two types of cores: customised for gaming/graphics rendering.

- Sony-Toshiba-IBM (STI) partnership built CELL processor was used in Sony's PlayStation 3 (2000-2005).

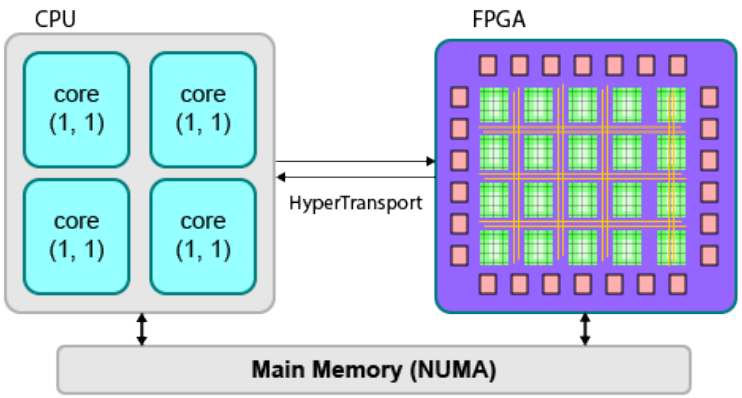


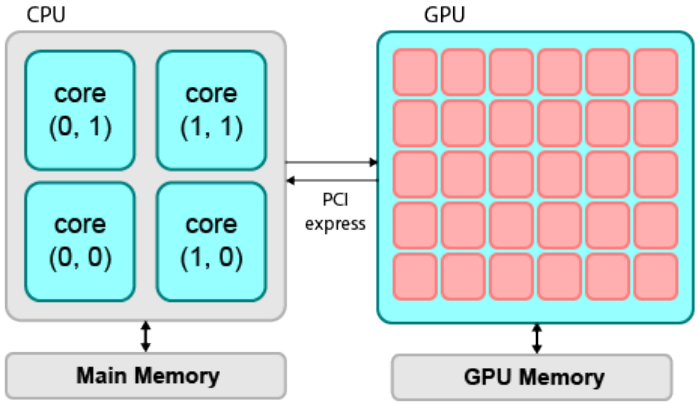
PPE: assign tasks to other cores; control communication between cores

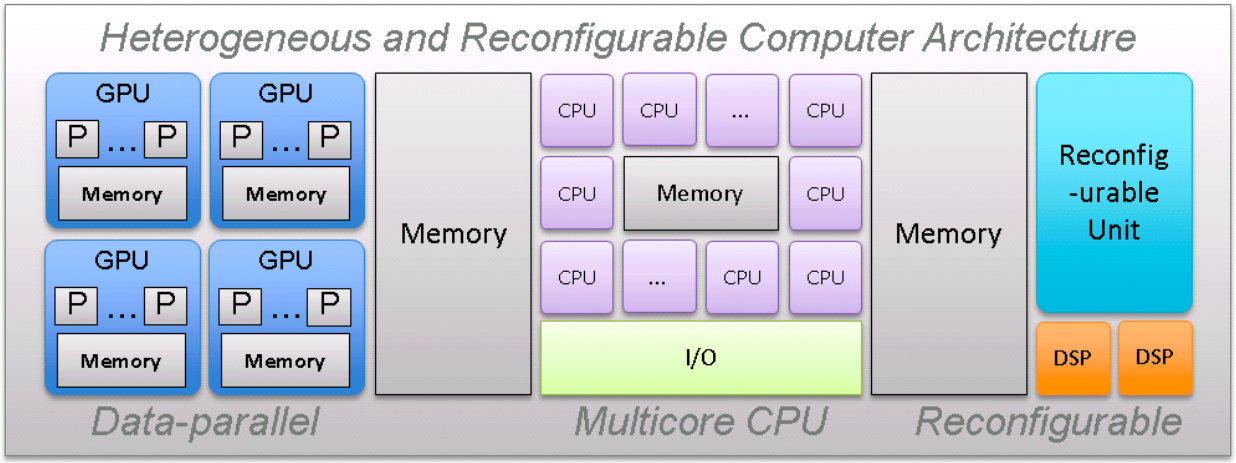
SPE: perform vector operations; provide high floating-point throughput

General Heterogeneous Computing Systems

- Heterogeneous systems can combine different processing technologies: FPGA, ASIC, GPP, GPU, DSP or microcontroller units to achieve desired performance with less power consumption.



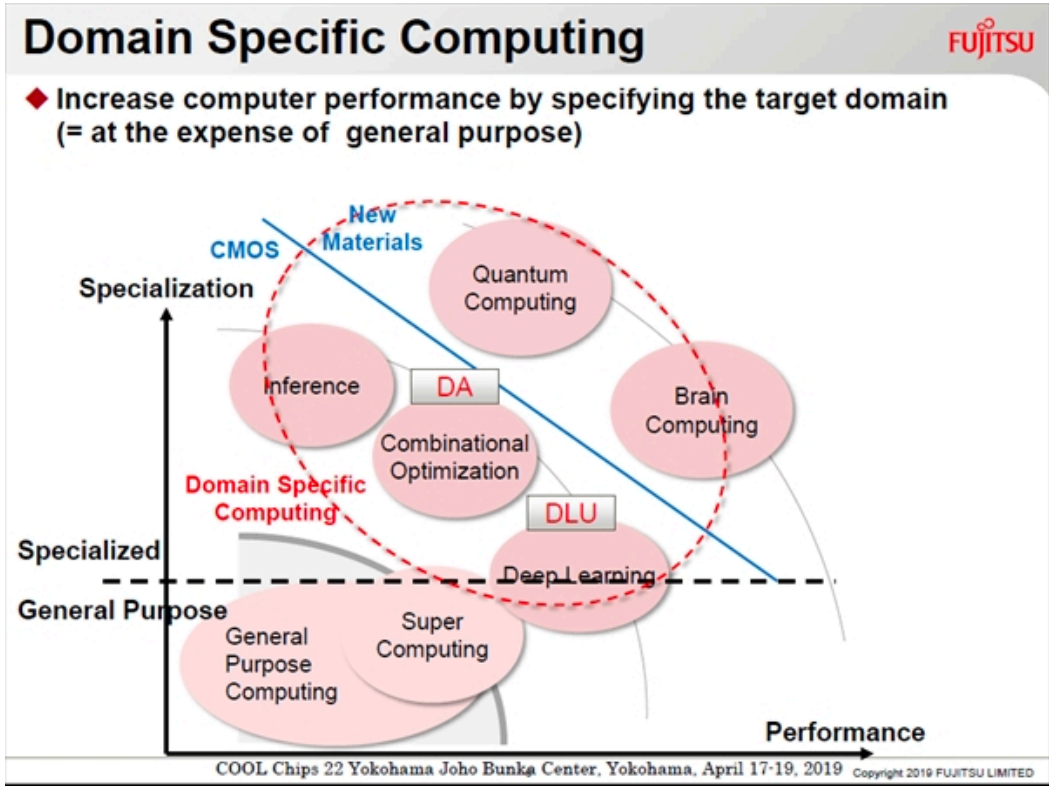




.....

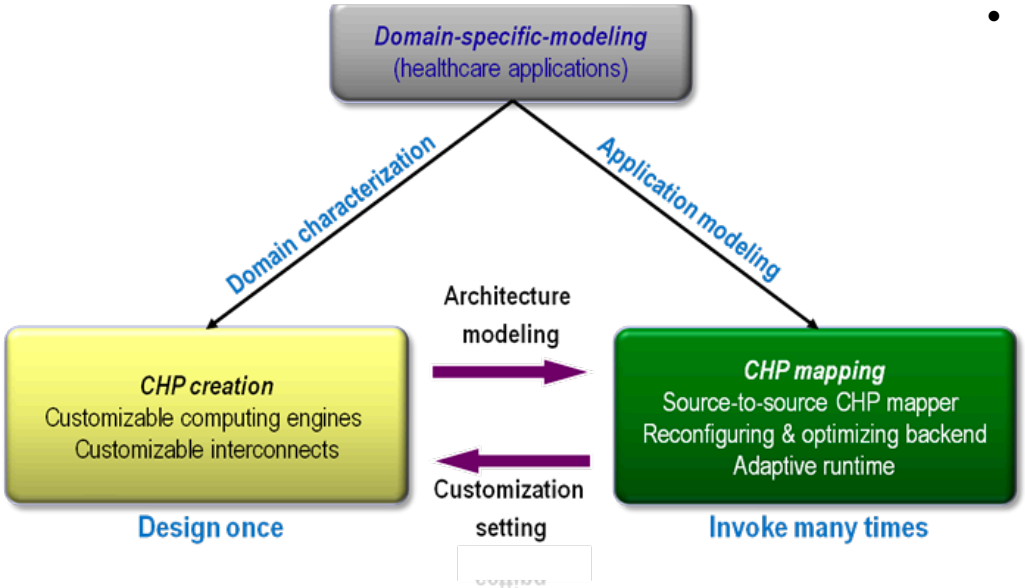
Domain Specific Computing (DSC)

- A domain specific computing system is a heterogeneous multicore system
- It is customised specifically according to the computations (the kind of processing) involved in a given application domain for high-performance and power efficient realisation of the computations for the particular application domain and support the changes in the application domain.
- A DSC system may contain some generic programmable core, customised cores, hardware accelerators and programmable interconnects.



DOMAIN SPECIFIC COMPUTING (DSC)

- Motivation:
  - Each user has a high computing demand in one or a few selected application domains
  - Customizable computing systems for particular app. domain for perf/Power efficiency
  - Make best use of techniques (ASIC, FPGA…) for power/perf efficiency with lower cost



- E.g., **healthcare application**, **graphics** for game developers, **circuit simulation** for integrated circuit design houses, **financial analytics** for investment banks, **transforms and filters** for signal/image and video processing, **matrix operations** for weather forecasting, scientific computing

- General computing needs like email, word processing, web browsing can be easily achieved by any existing computing technology

Domain Specific Computing (DSC)

- Customizable heterogeneous platform of a DSC system
  - Fixed cores: differ widely in terms of energy efficiency, performance and area

    -- They have very limited (or no) re-configurability

    -- Allow voltage/frequency scaling to adapt power/performance
  - Customizable cores: architecture customised (tuned) according to application

    -- Offer a set of tunable options (e.g., register file sizes, cache sizes, bit-width of datapath operating frequency and supply voltages)
  - Programmable fabrics: provide maximum flexibility

    -- Implement custom instructions by specialised co-processor (FFT, DCT, filters, etc.) to accelerate performance

    -- Architecture customised in terms of # computing units, types of computing units, # pipeline stages, etc. to implement complex operations

Processor Technology Summary

- Programmable Processors
  - General purpose processor
    -- Single-core and multicore processors
  - Specialised programmable processor
    -- Application Specific Instruction Set Processors (ASIP), GPU, DSP
- Dedicated Processors
  - Reconfigurable processors (e.g., FPGA)
  - Application Specific Integrated Circuits (ASIC)
- Heterogeneous System On Chip (SoC)
  - ARM big. LITTLE
  - IBM Cell Broadband Engine (BE) architecture

Choice of Implementation Platforms

- System cost; Speed; Power consumption; Design reuse; Time-to-Market

Final Exam Mock

1. Assume you have an in-order RISC processor that has a CPI of 1.3 with an ideal memory hierarchy (i.e., all memory accesses hit in the L1 caches). For a given benchmark, 38% of instructions executed are loads and stores. The miss penalty for reads and writes to the L1 data cache is 20 clock cycles and the miss rate is 3%.
    - Calculate CPI of the RISC processor
    - Memory access latency
    - Cache hate rate
    - Cache miss rate

Final Exam Mock

2. Parallelisms discussed in the lectures:
   - Types of parallelisms;
   - Three major tasks required for parallel executions of the instructions (Instruction Level Parallelism);
   - Methods to extract more Instruction-Level Parallelism.

3. Pipeline harzards:
   - List the three types of Pipeline harzards;
   - Describe the main reason(s) respectively causing the pipeline harzards.

Final Exam Mock

4. Answer the qualitative questions about the branch prediction discussed in the lectures:
   - The approaches to deal with branch harzards/control harzards;
   - Static Branch Prediction and Dynamic Branch Prediction;
   - The function of a Branch Target Buffer (BTB) and Branch History Table (BHT).

5. Give a brief overview about the future of computing discussed in class
   - The custom and emerging computing trends of computing architectures

     (e.g., GPP, DSP, GPU, FPGA and ASIC)
   - Make a comparison.

FINAL EXAM MOCK

6. Analyze the following code segment adds a scalar to a vector:

$$for(i = 64;\ \ i > 0;\ \ i = i - 1)$$
$$x[i] = x[i] + s;$$

(a) (5 points) Can the loop iterations above be parallelized? Why or why not?

(b) (5 points) When the code above is compiled, the following MIPS assembly is produced.

```
loop:  l.d     $f0, 0($t0)      #f0 = array element;
       add.d   $f4, $f0, $f2    #add scalar in f2;
       s.d     $f4, 0($t0)      #store result;
       addi    $t0, $t0, -8     #decrement pointer (8 bytes DW);
       bne     $t0, $t1, Loop   #branch if t0! = t1.
```

Identify all of the RAW hazards in the assembly program above.